Design of Digital Circuits

Lecture 25: Virtual Memory II

Prof. Onur Mutlu

ETH Zurich

Spring 2019

24 May 2019

Readings

- Virtual Memory

- Required
  - H&H Chapter 8.4

Recall: Virtual Memory

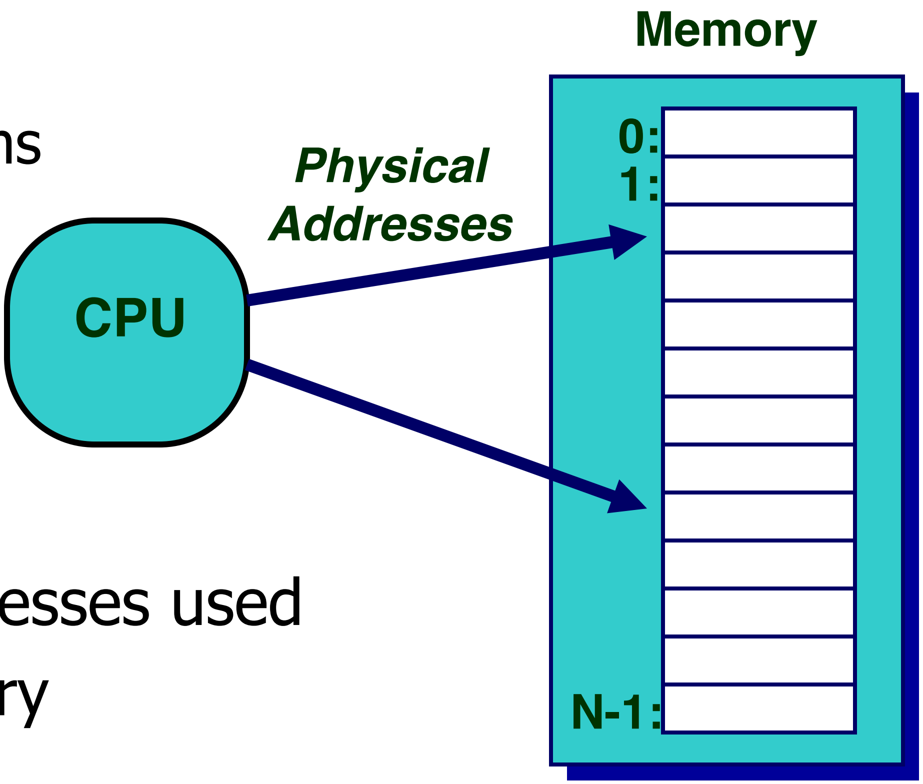
- Idea: Give the programmer the illusion of a large address space while having a small physical memory
  - So that the programmer does not worry about managing physical memory

- Programmer can assume he/she has “infinite” amount of physical memory

- Hardware and software cooperatively and automatically manage the physical memory space to provide the illusion
  - Illusion is maintained for each independent process

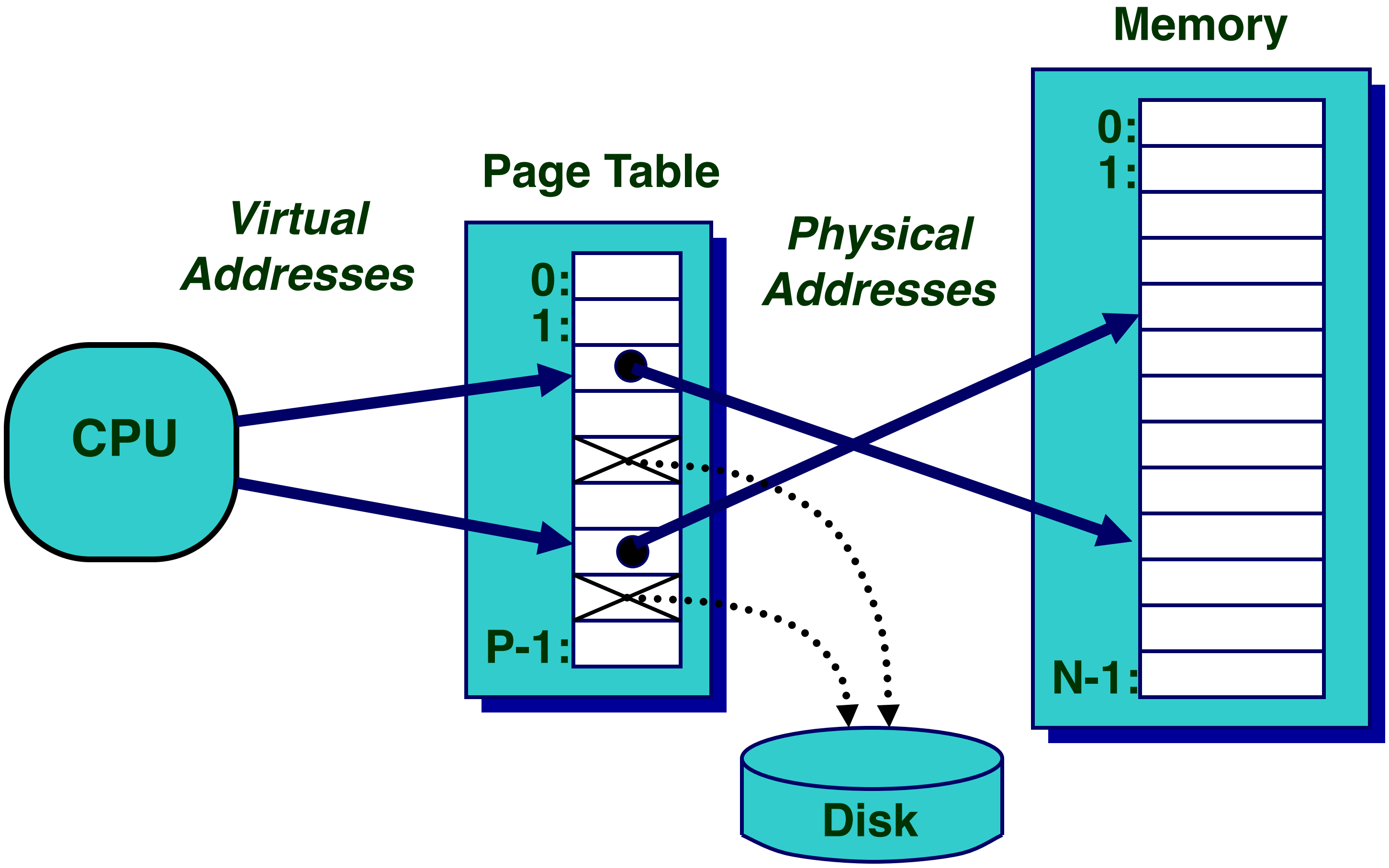
Recall: A System with Physical Memory Only

- Examples:
  - most Cray machines
  - early PCs
  - many embedded systems



CPU's load or store addresses used directly to access memory

A System with Virtual Memory (Page based)



- Address Translation: The hardware converts virtual addresses into physical addresses via an OS-managed lookup table (page table)

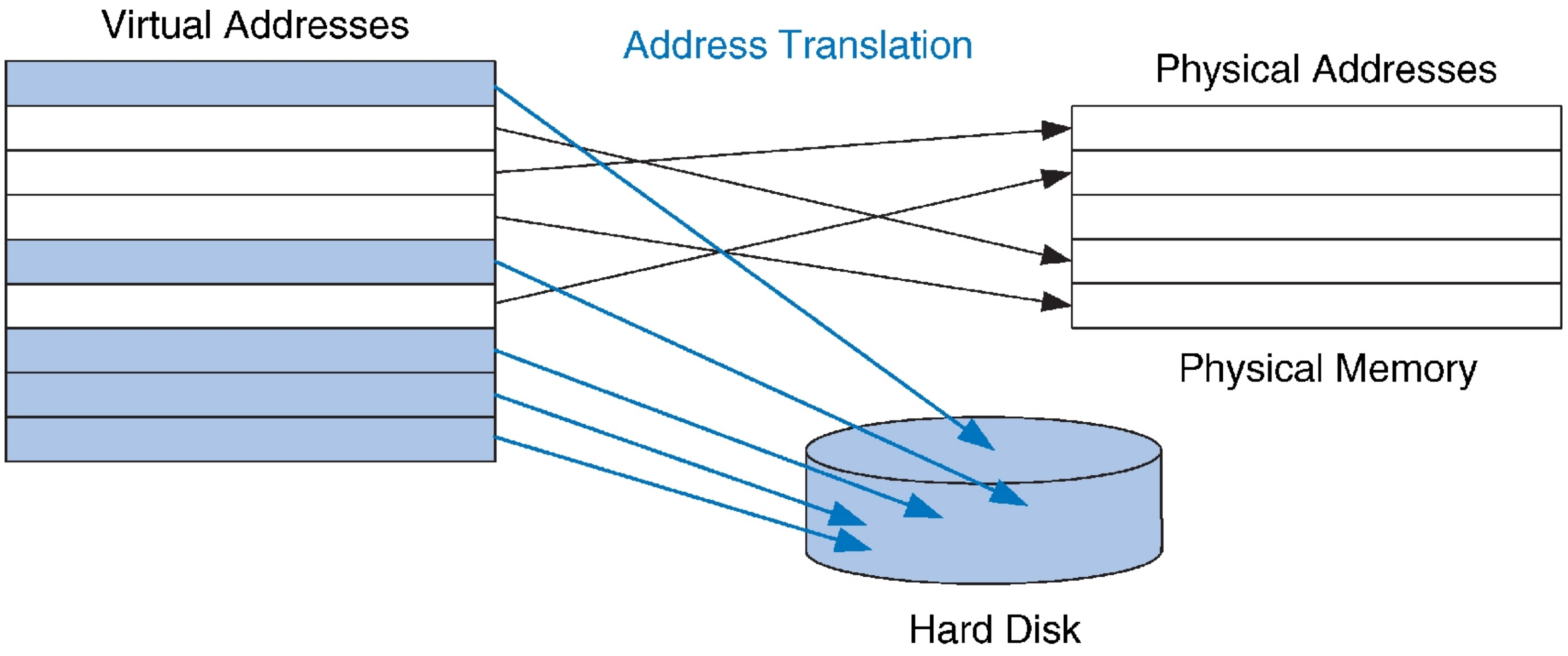
Recall: Virtual Memory Definitions

- Page size: amount of memory transferred from hard disk to DRAM at once

- Address translation: determining the physical address from the virtual address

- Page table: lookup table used to translate virtual addresses to physical addresses (and find where the associated data is)

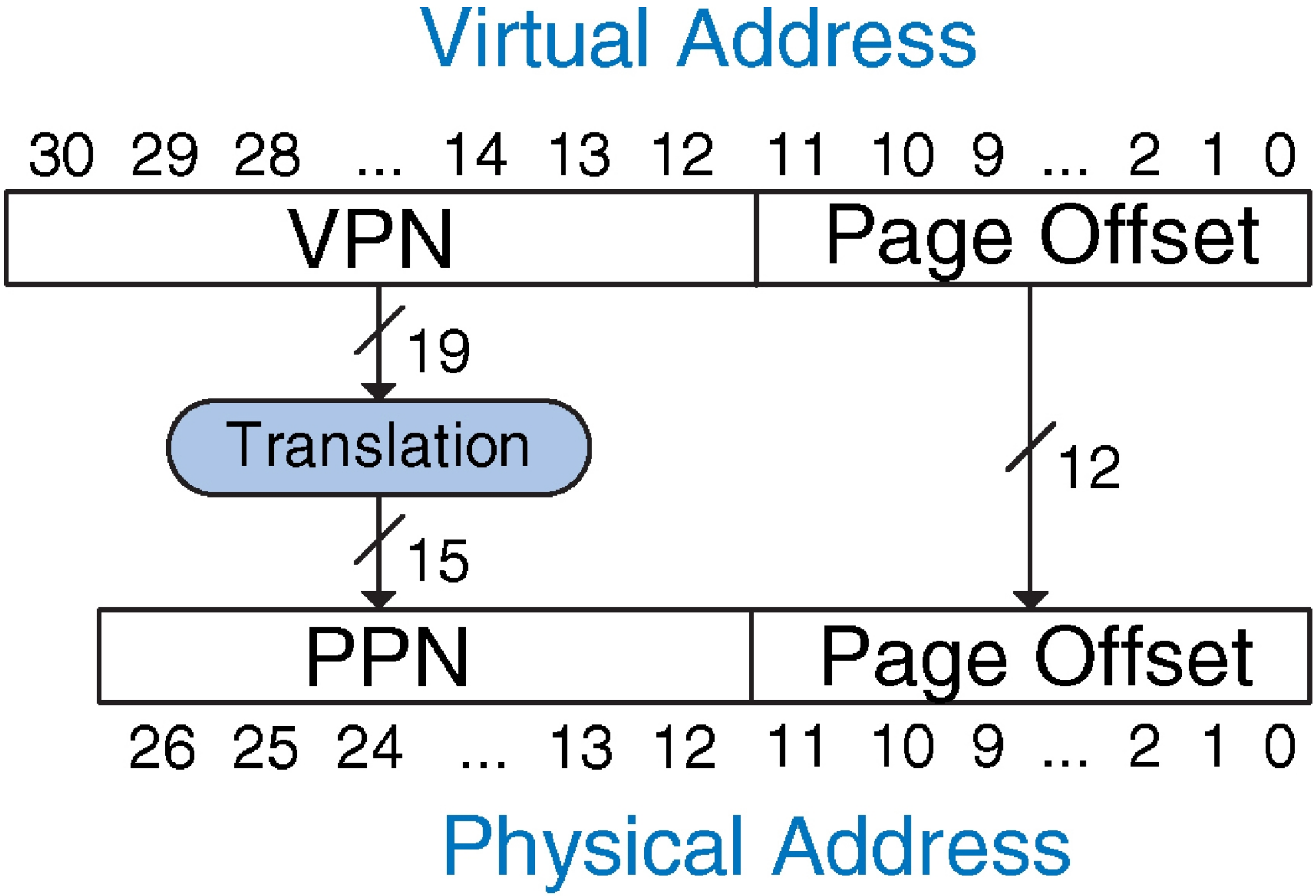
Recall: Virtual and Physical Addresses



© 2007 Elsevier, Inc. All rights reserved

- Most accesses hit in physical memory
- But programs see the large capacity of virtual memory

Recall: Address Translation



© 2007 Elsevier, Inc. All rights reserved

Recall: Virtual Memory Example

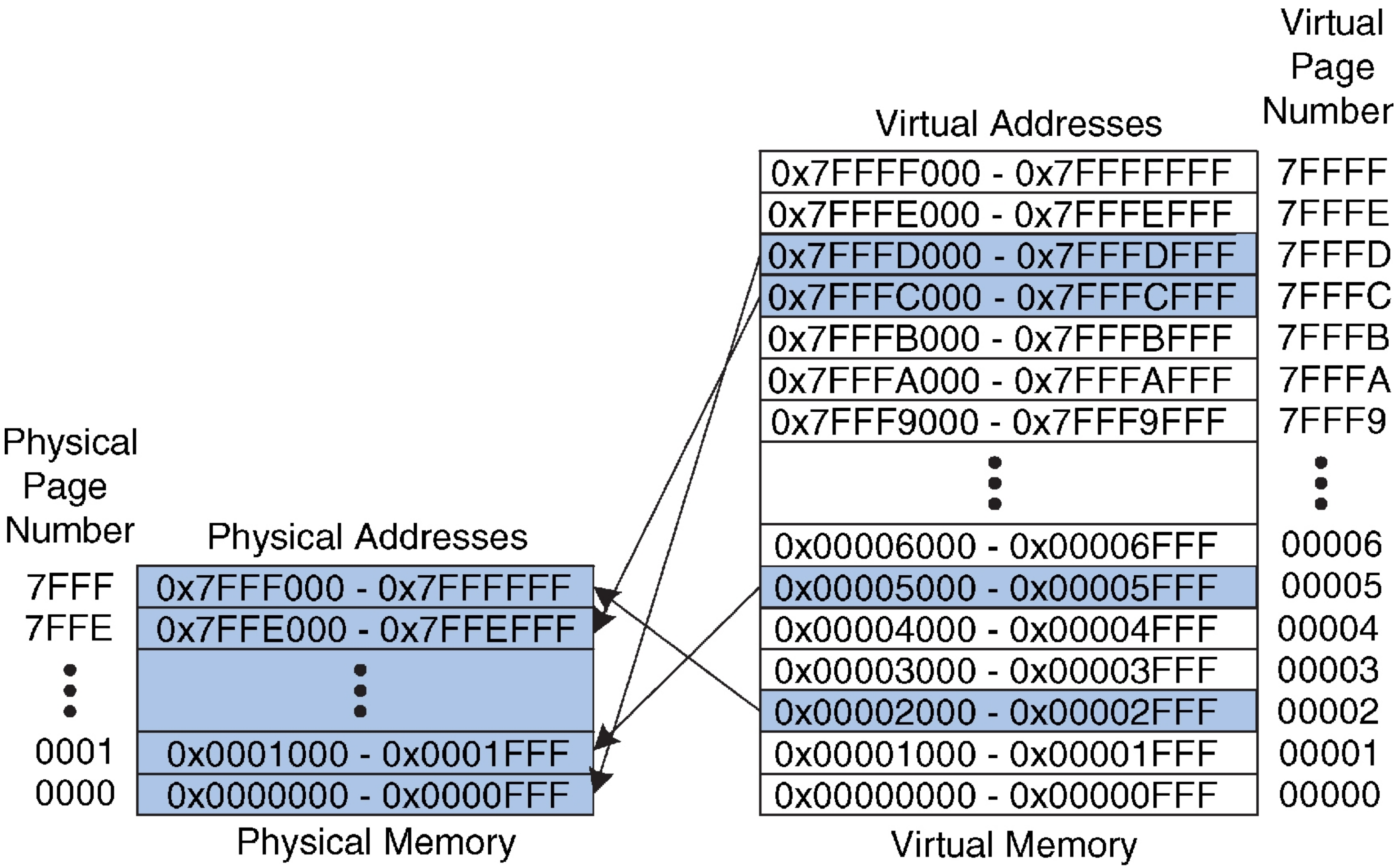
- System:
  - Virtual memory size: 2 GB = 2^{31} bytes
  - Physical memory size: 128 MB = 2^{27} bytes
  - Page size: 4 KB = 2^{12} bytes

Recall: Virtual Memory Example

- System:
  - Virtual memory size: 2 GB = 2^{31} bytes
  - Physical memory size: 128 MB = 2^{27} bytes
  - Page size: 4 KB = 2^{12} bytes

- Organization:
  - Virtual address: **31** bits
  - Physical address: **27** bits
  - Page offset: **12** bits
  - # Virtual pages = $2^{31}/2^{12}$ = $\mathbf{2^{19}}$ (VPN = 19 bits)
  - # Physical pages = $2^{27}/2^{12}$ = $\mathbf{2^{15}}$ (PPN = 15 bits)

Recall: Virtual Memory Mapping Example



Physical Page Number	Physical Addresses
7FFF	0x7FFF000 - 0x7FFFFFF
7FFE	0x7FFE000 - 0x7FFEFFF
⋮	⋮
0001	0x0001000 - 0x0001FFF
0000	0x0000000 - 0x0000FFF

Physical Memory

Virtual Addresses	Virtual Page Number
0x7FFFF000 - 0x7FFFFFFF	7FFFF
0x7FFFE000 - 0x7FFFEFFF	7FFFE
0x7FFFD000 - 0x7FFFDFFF	7FFFD
0x7FFFC000 - 0x7FFFCFFF	7FFFC
0x7FFFB000 - 0x7FFFBFFF	7FFFB
0x7FFFA000 - 0x7FFFAFFF	7FFFA
0x7FFF9000 - 0x7FFF9FFF	7FFF9
⋮	⋮
0x00006000 - 0x00006FFF	00006
0x00005000 - 0x00005FFF	00005
0x00004000 - 0x00004FFF	00004
0x00003000 - 0x00003FFF	00003
0x00002000 - 0x00002FFF	00002
0x00001000 - 0x00001FFF	00001
0x00000000 - 0x00000FFF	00000

Virtual Memory

How Do We Translate Addresses?

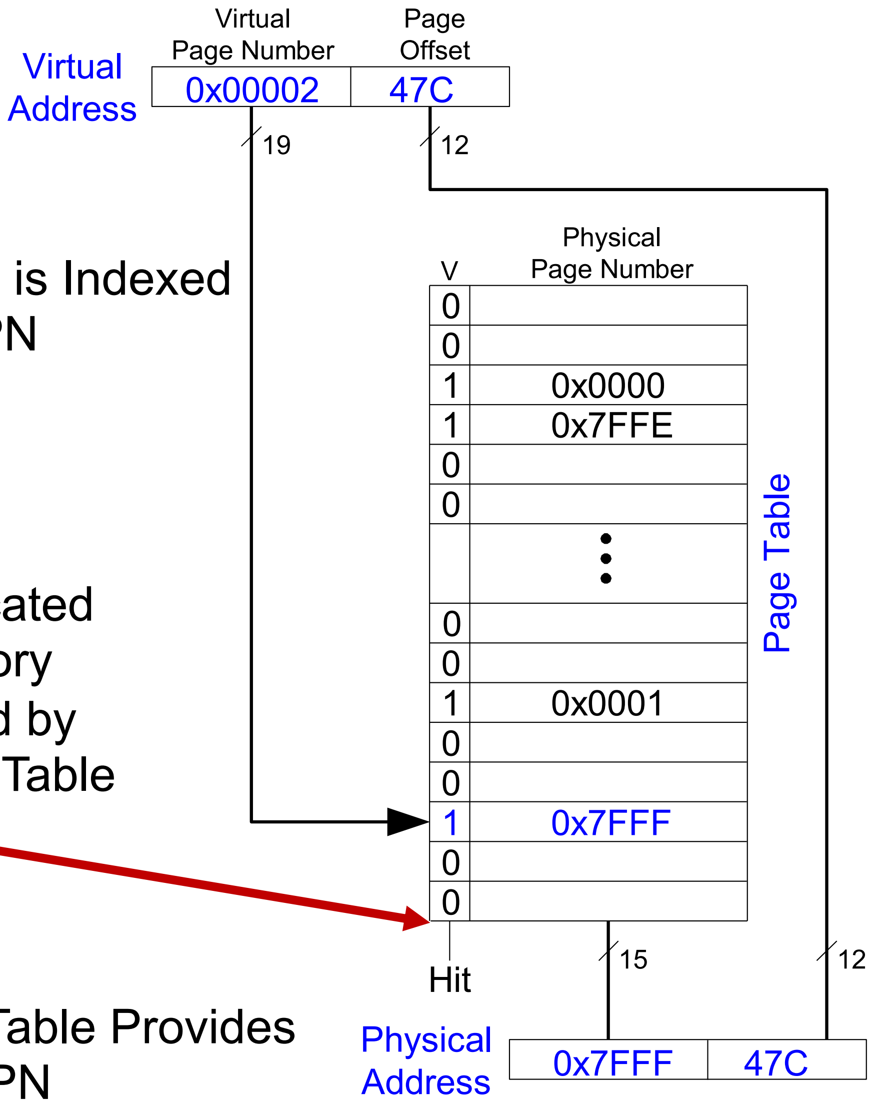
- **Page table**
  - Has entry for each virtual page

- Each page table entry has:
  - Valid bit: whether the virtual page is located in physical memory (if not, it must be fetched from the hard disk)
  - Physical page number: where the virtual page is located in physical memory
  - (Replacement policy, dirty bits)

Page Table Address Translation Example

Page Table is Indexed with the VPN

Page Table is located at physical memory address specified by the PTBR (Page Table Base Register)



V	Physical Page Number
0	
0	
1	0x0000
1	0x7FFE
0	
0	
	⋮
0	
0	
1	0x0001
0	
0	
1	0x7FFF
0	
0	

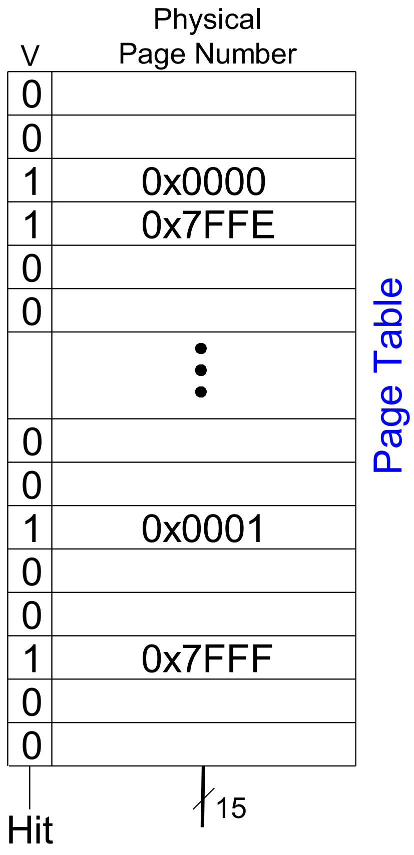
Page Table Provides The PPN

Page offset bits do not change during translation

Page Table Address Translation Example 1

- What is the physical address of virtual address 0x5F20?

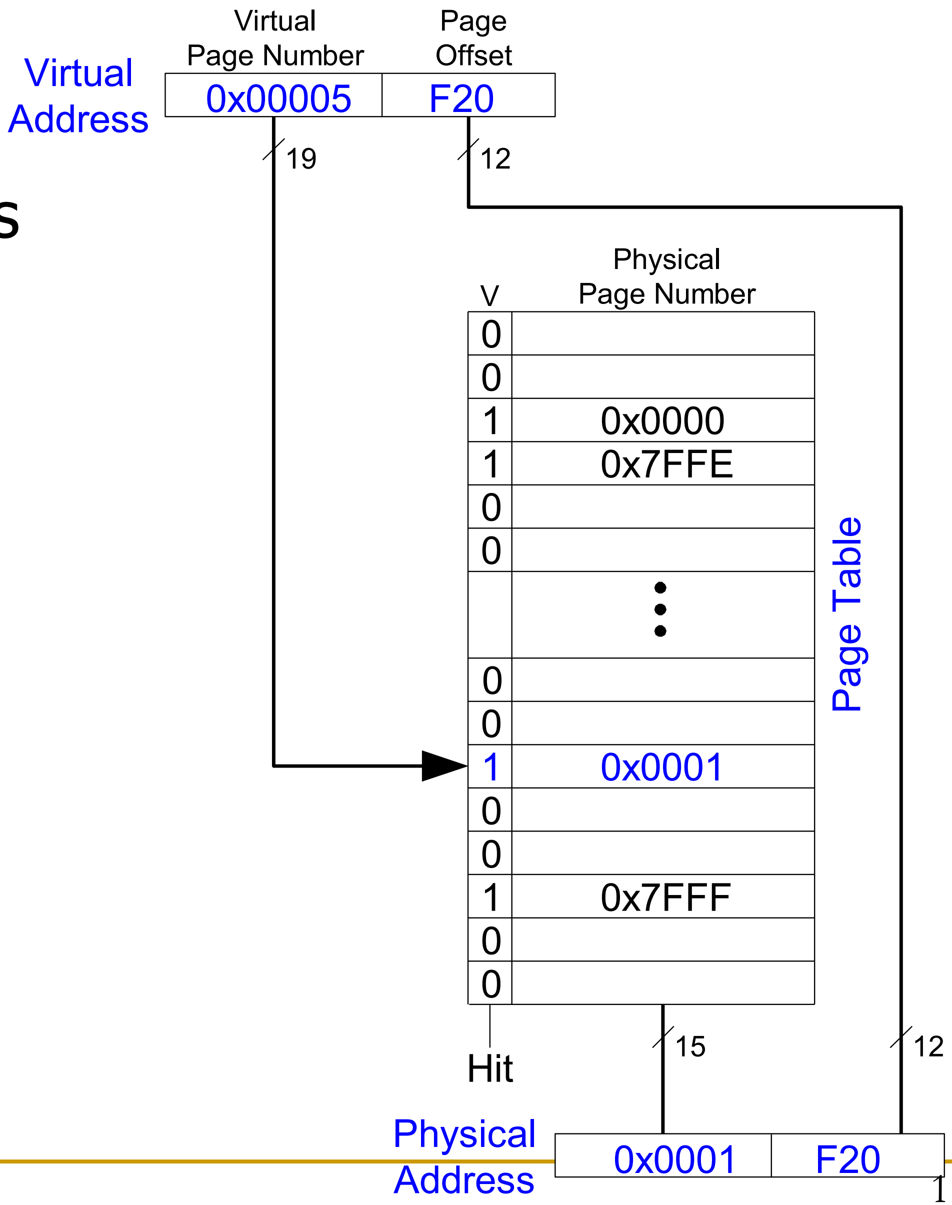
- We first need to find the page table entry containing the translation for the corresponding VPN
- Look up the PTE at the address
  - PTBR + VPN*PTE-size



V	Physical Page Number
0	
0	
1	0x0000
1	0x7FFE
0	
0	
	⋮
0	
0	
1	0x0001
0	
0	
1	0x7FFF
0	
0	

Page Table Address Translation Example 1

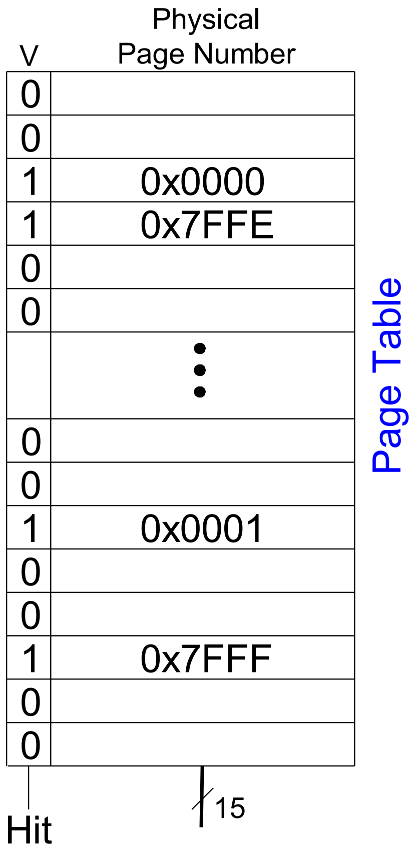
- What is the physical address of virtual address 0x5F20?
  - VPN = 5
  - Entry 5 in page table indicates VPN 5 is in physical page 1
  - Physical address is 0x1F20



V	Physical Page Number
0	
0	
1	0x0000
1	0x7FFE
0	
0	
	⋮
0	
0	
1	0x0001
0	
0	
1	0x7FFF
0	
0	

Page Table Address Translation Example 2

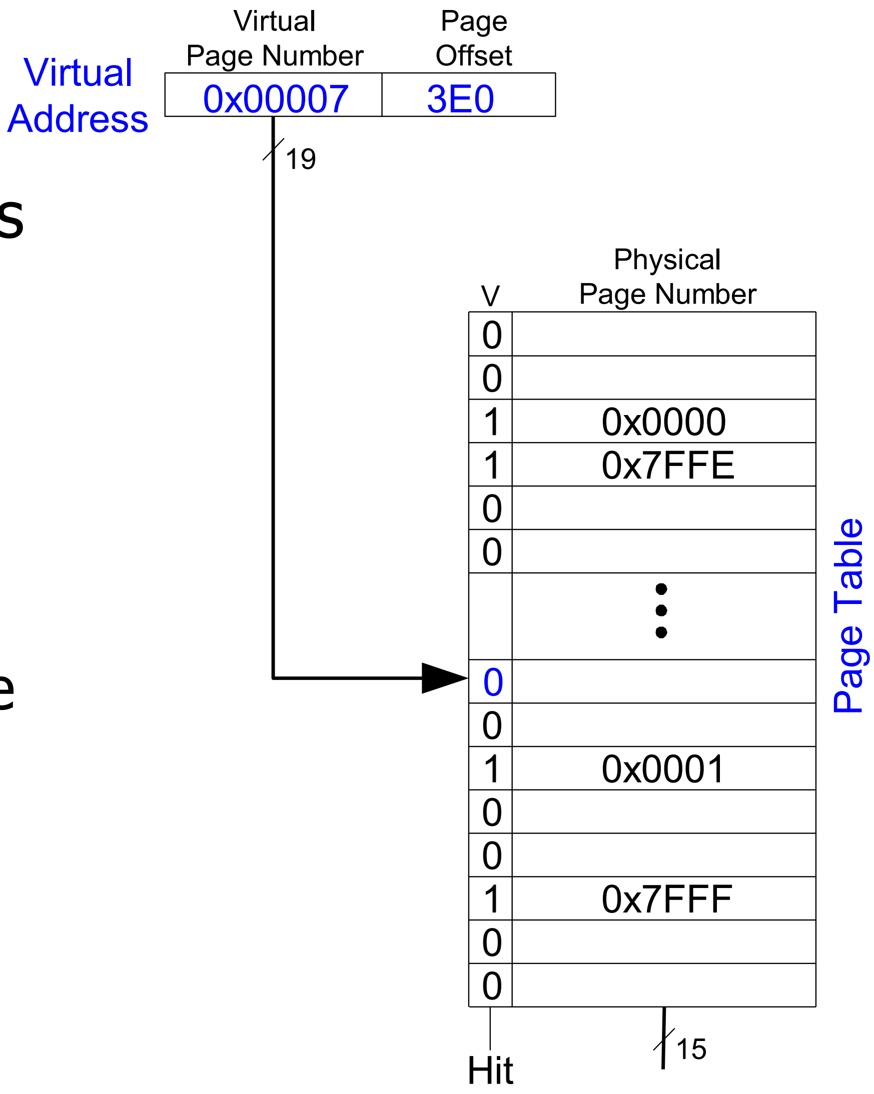
- What is the physical address of virtual address 0x73E0?



V	Physical Page Number
0	
0	
1	0x0000
1	0x7FFE
0	
0	
	⋮
0	
0	
1	0x0001
0	
0	
1	0x7FFF
0	
0	

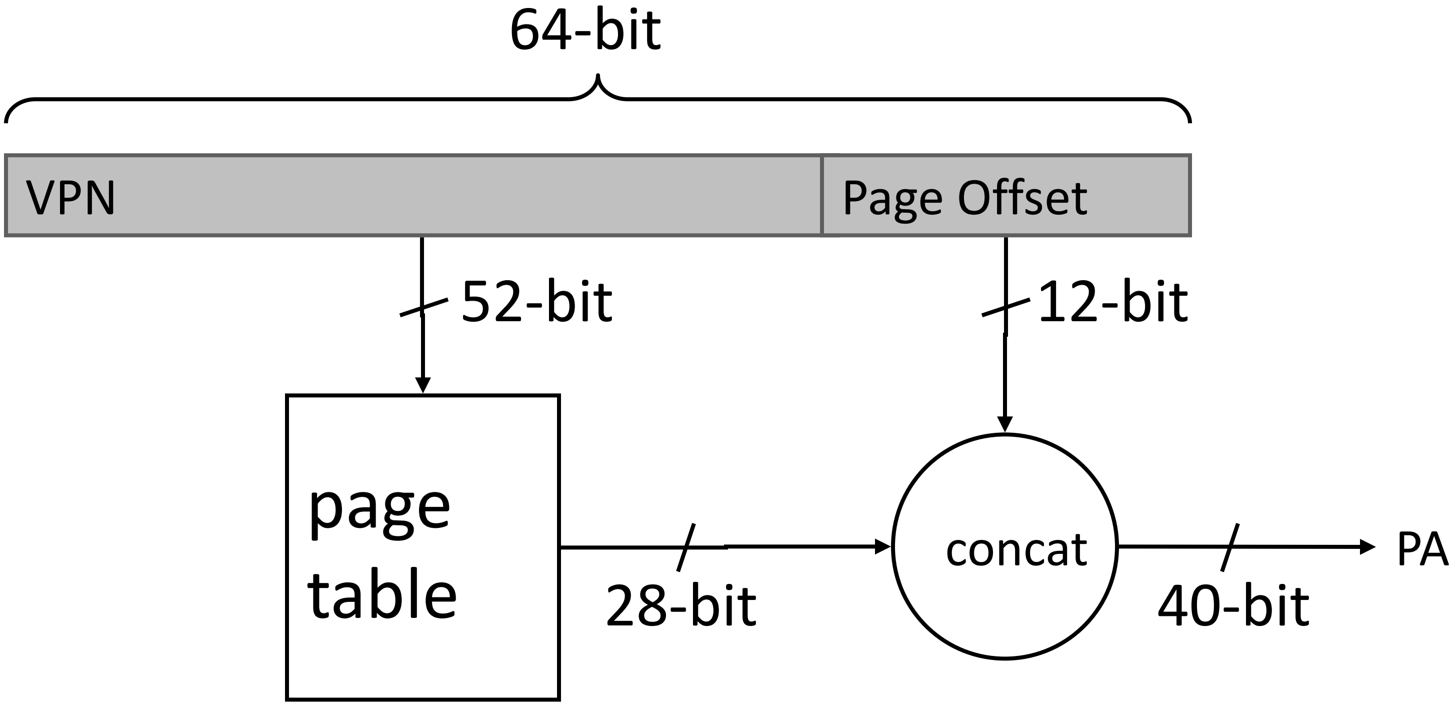
Page Table Address Translation Example 2

- What is the physical address of virtual address 0x73E0?
  - VPN = 7
  - Entry 7 in page table is invalid, so the page is not in physical memory
  - The virtual page must be swapped into physical memory from disk



V	Physical Page Number
0	
0	
1	0x0000
1	0x7FFE
0	
0	
	⋮
0	
0	
1	0x0001
0	
0	
1	0x7FFF
0	
0	

Issue: Page Table Size



- Suppose 64-bit VA and 40-bit PA, how large is the page table?
  - **2^{52} entries x ~4 bytes ≈ 2^{54} bytes**

  and that is for just one process!

  and the process may not be using the entire VM space!

Page Table Challenges

- Challenge 1: Page table is large
  - at least part of it needs to be located in physical memory
  - solution: multi-level (hierarchical) page tables

- Challenge 2: Each instruction fetch or load/store requires at least two memory accesses:

  1. one for address translation (page table read)
  2. one to access data with the physical address (after translation)

- Two memory accesses to service an instruction fetch or load/store greatly degrades execution time
  - Unless we are clever… → speed up the translation…

Translation Lookaside Buffer (TLB)

- Idea: Cache the page table entries (PTEs) in a hardware structure in the processor to speed up address translation

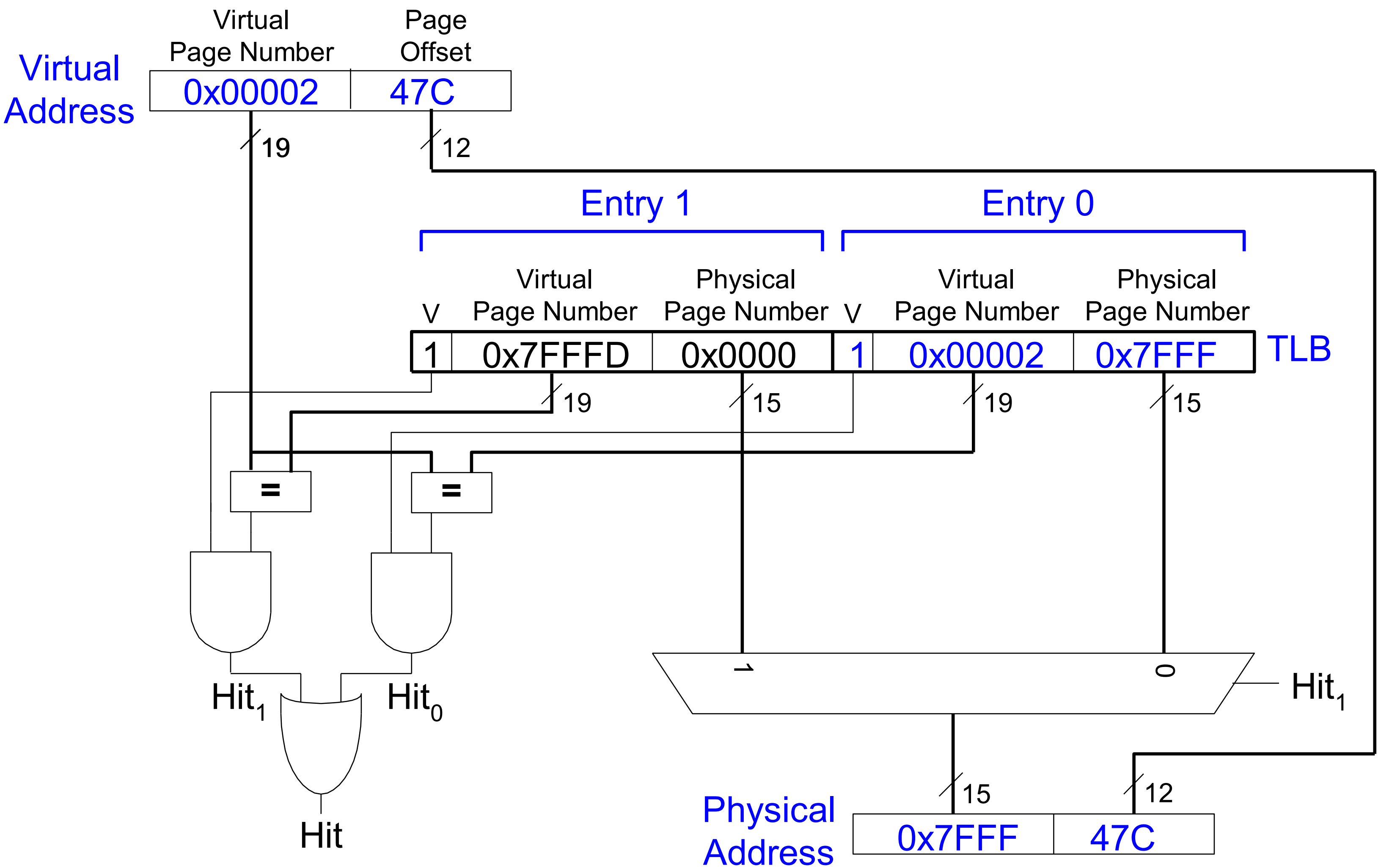
- Translation lookaside buffer (TLB)
  - Small cache of most recently used translations (PTEs)
  - Reduces number of memory accesses required for *most* instruction fetches and loads/stores to only one

Translation Lookaside Buffer (TLB)

- Page table accesses have a lot of temporal locality
  - Data accesses have temporal and spatial locality
  - Large page size (say 4KB, 8KB, or even 1-2GB)
  - Consecutive instructions and loads/stores are likely to access same page

- TLB
  - Small: accessed in ~ 1 cycle
  - Typically 16 - 512 entries
  - High associativity
  - > 95-99 % hit rates typical (depends on workload)
  - Reduces number of memory accesses for most instruction fetches and loads/stores to only one

Example Two-Entry TLB



Virtual Memory Support and Examples

Supporting Virtual Memory

- Virtual memory requires both HW+SW support
  - Page Table is in memory
  - Can be cached in special hardware structures called Translation Lookaside Buffers (TLBs)

- The hardware component is called the MMU (memory management unit)
  - Includes Page Table Base Register(s), TLBs, page walkers

- It is the job of the software to leverage the MMU to
  - Populate page tables, decide what to replace in physical memory
  - Change the Page Table Register on context switch (to use the running thread's page table)
  - Handle page faults and ensure correct mapping

Address Translation

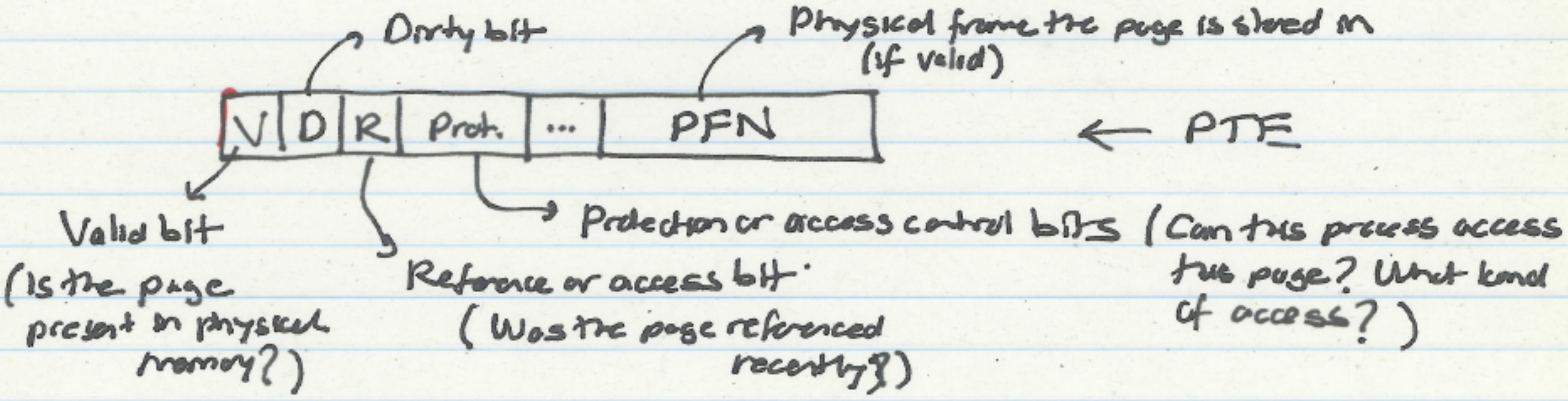
- How to obtain the physical address from a virtual address?

- Page size specified by the ISA
  - VAX: 512 bytes
  - Today: 4KB, 8KB, 2GB, … (small and large pages mixed together)
  - Trade-offs? (remember cache lectures)

- Page Table contains an entry for each virtual page
  - Called Page Table Entry (PTE)
  - What is in a PTE?

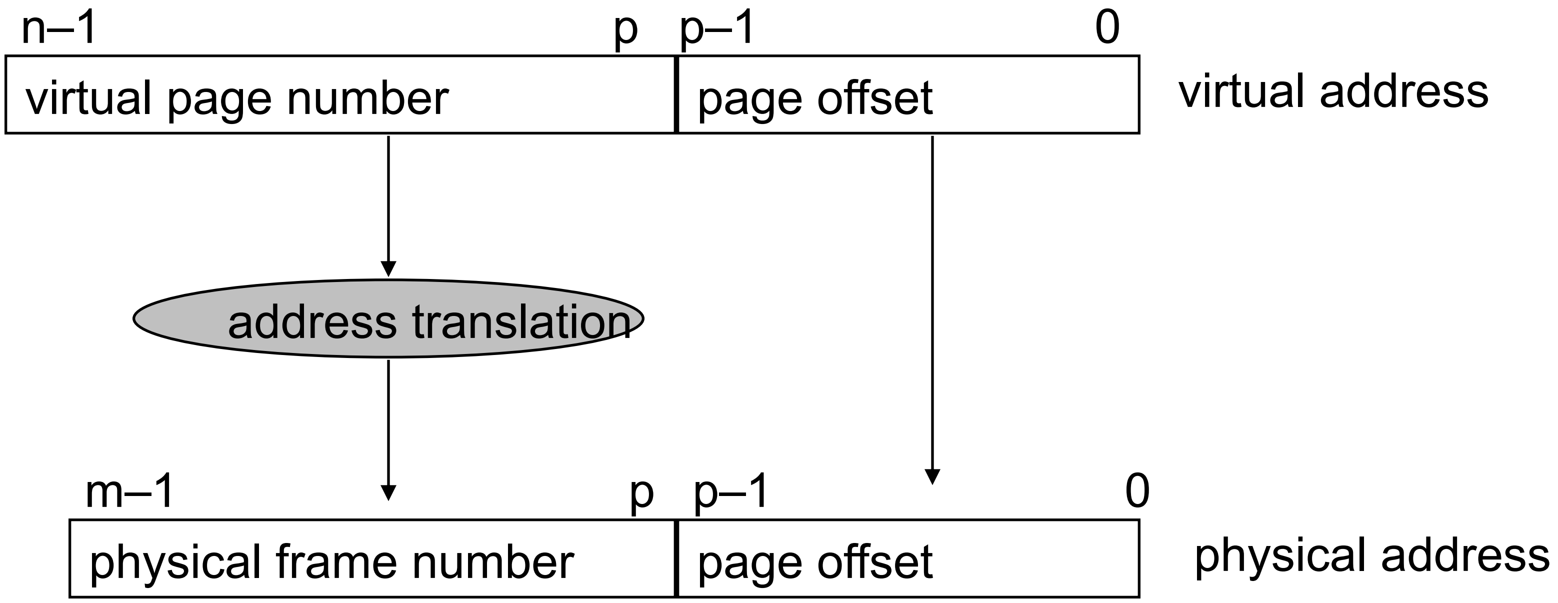
What Is in a Page Table Entry (PTE)?

- Page table is the "tag store" for the physical memory data store
  - A mapping table between virtual memory and physical memory
- PTE is the "tag store entry" for a virtual page in memory
  - Need a valid bit → to indicate validity/presence in physical memory
  - Need tag bits (PFN) → to support translation
  - Need bits to support replacement
  - Need a dirty bit to support "write back caching"
  - Need protection bits to enable access control and protection


Dirty bit
Physical frame the page is stored in (if valid)
V
D
R
Prot.
...
PFN
← PTE
Valid bit (Is the page present in physical memory?)
Reference or access bit. (Was the page referenced recently?)
Protection or access control bits (Can this process access this page? What kind of access?)

Address Translation (I)

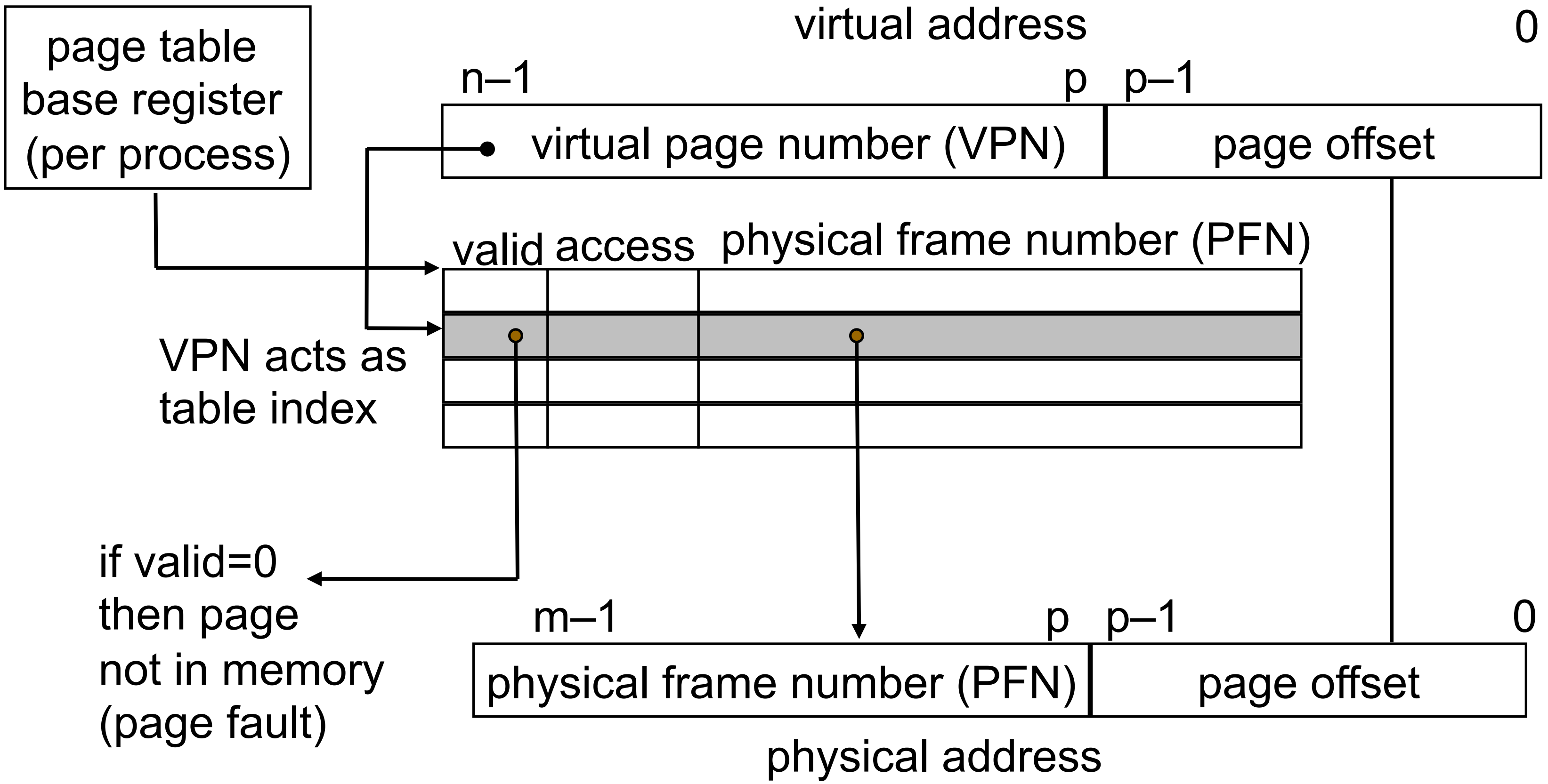
- Parameters
  - $P = 2^p$ = page size (bytes).
  - $N = 2^n$ = Virtual-address limit
  - $M = 2^m$ = Physical-address limit



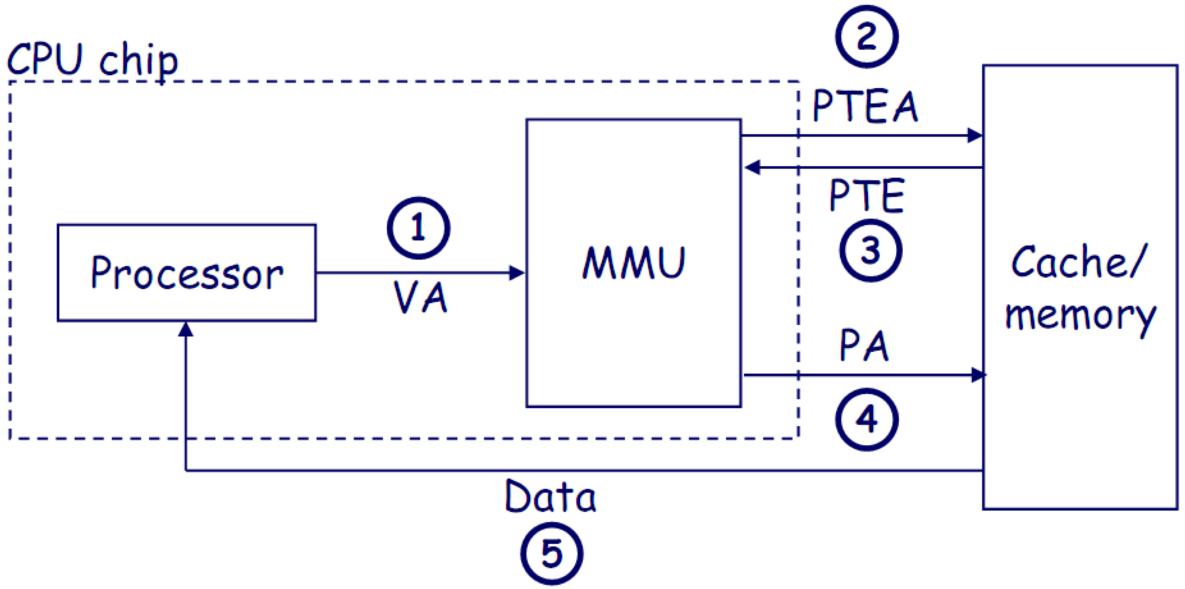
Page offset bits don’t change as a result of translation

Address Translation (II)

- Separate (set of) page table(s) per process
- VPN forms index into page table (points to a page table entry)
- Page Table Entry (PTE) provides information about page


page table base register (per process)
virtual address
n–1
p
p–1
0
virtual page number (VPN)
page offset
valid
access
physical frame number (PFN)
VPN acts as table index
if valid=0 then page not in memory (page fault)
m–1
p
p–1
0
physical frame number (PFN)
page offset
physical address

Address Translation: Page Hit



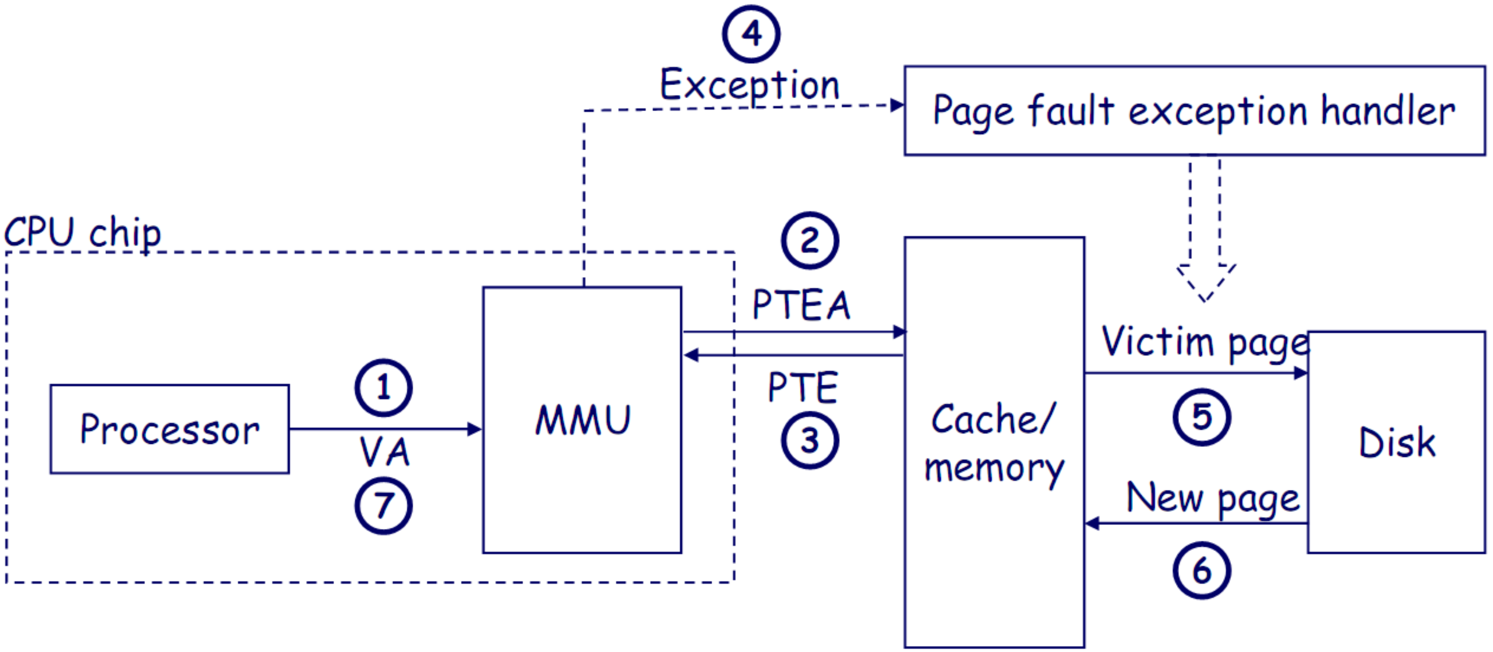
1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in memory

4) MMU sends physical address to L1 cache

5) L1 cache sends data word to processor

Address Translation: Page Fault



1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in memory

4) Valid bit is zero, so MMU triggers page fault exception

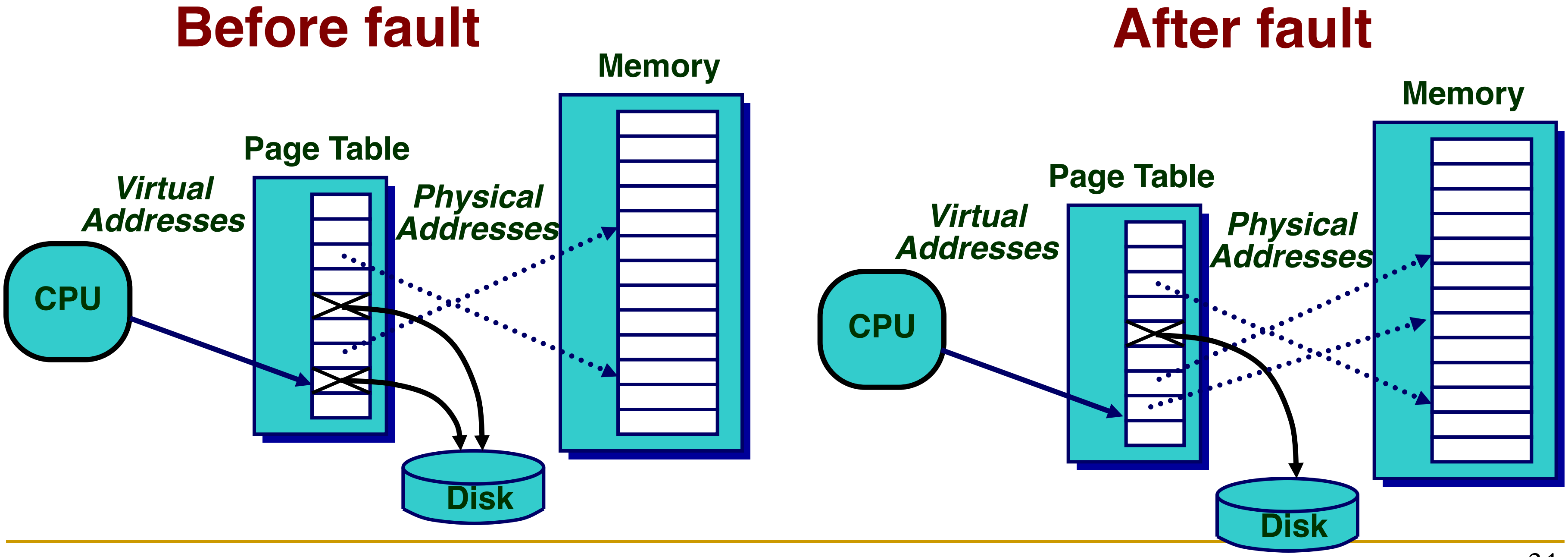
5) Handler identifies victim, and if dirty pages it out to disk

6) Handler pages in new page and updates PTE in memory

7) Handler returns to original process, restarting faulting instruction.

Page Fault ("A Miss in Physical Memory")

- If a page is not in physical memory but disk
  - Page table entry indicates virtual page not in memory
  - Access to such a page triggers a page fault exception
  - OS trap handler invoked to move data from disk into memory
    - Other processes can continue executing
    - OS has full control over placement

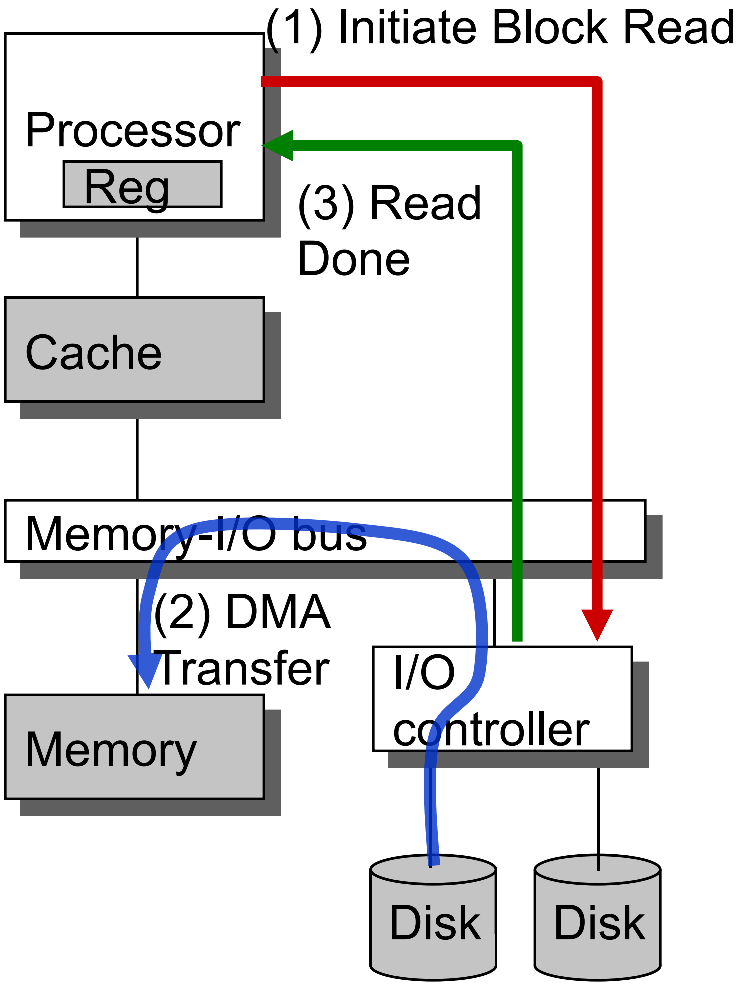


Before fault

After fault

Servicing a Page Fault

- (1) Processor signals controller
  - Read block of length P starting at disk address X and store starting at memory address Y
- (2) Read occurs
  - Direct Memory Access (DMA)
  - Under control of I/O controller
- (3) Controller signals completion
  - Interrupt processor
  - OS resumes suspended process



Page Replacement Algorithms

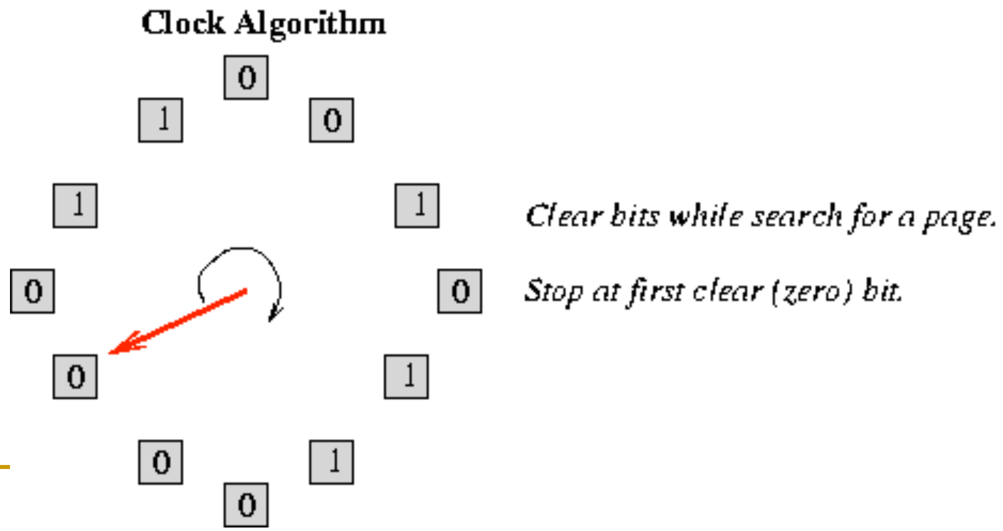
- If physical memory is full (i.e., list of free physical pages is empty), which physical frame to replace on a page fault?

- Is True LRU feasible?
  - 4GB memory, 4KB pages, how many possibilities of ordering?

- Modern systems use approximations of LRU
  - E.g., the CLOCK algorithm
- And, more sophisticated algorithms to take into account "frequency" of use
  - E.g., the ARC algorithm
  - Megiddo and Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," FAST 2003.

CLOCK Page Replacement Algorithm

- Keep a circular list of physical frames in memory (OS does)
- Keep a pointer (hand) to the last-examined frame in the list
- When a page is accessed, set the R bit in the PTE
- When a frame needs to be replaced, replace the first frame that has the reference (R) bit not set, traversing the circular list starting from the pointer (hand) clockwise
  - During traversal, clear the R bits of examined frames
  - Set the hand pointer to the next frame in the list


Clock Algorithm

Clear bits while search for a page.

Stop at first clear (zero) bit.

Cache versus Page Replacement

- Physical memory (DRAM) is a cache for disk
  - Managed by system software via the virtual memory subsystem

- Page replacement is similar to cache replacement
- Page table is the "tag store" for physical memory data store

- What is the difference?
  - Required speed of access to cache vs. physical memory
  - Number of blocks in a cache vs. physical memory
  - "Tolerable" amount of time to find a replacement candidate (disk versus memory access latency)
  - Role of hardware versus software

Memory Protection

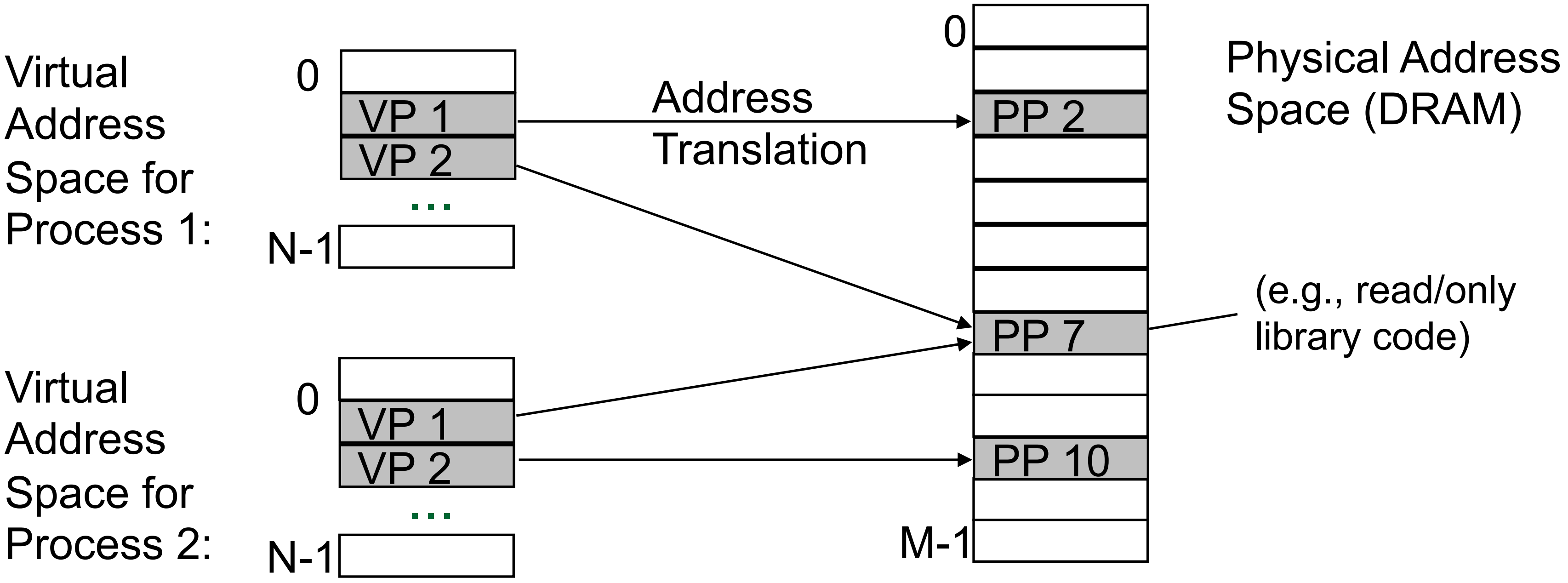
Memory Protection

- Multiple programs (*processes*) run at once
  - Each process has its own page table
  - Each process can use entire virtual address space without worrying about where other programs are

- A process can only access physical pages mapped in its page table – cannot overwrite memory of another process
  - Provides protection and isolation between processes
  - Enables access control mechanisms per page

Page Table is Per Process

- Each process has its own virtual address space
  - Full address space for each program
  - Simplifies memory allocation, sharing, linking and loading.


Virtual Address Space for Process 1:
0
VP 1
VP 2
...
N-1
Address Translation
0
PP 2
PP 7
PP 10
M-1
Physical Address Space (DRAM)
(e.g., read/only library code)
Virtual Address Space for Process 2:
0
VP 1
VP 2
...
N-1

Access Protection/Control via Virtual Memory

Page-Level Access Control (Protection)

- Not every process is allowed to access every page
  - E.g., may need supervisor level privilege to access system pages

- Idea: Store access control information on a page basis in the process's page table

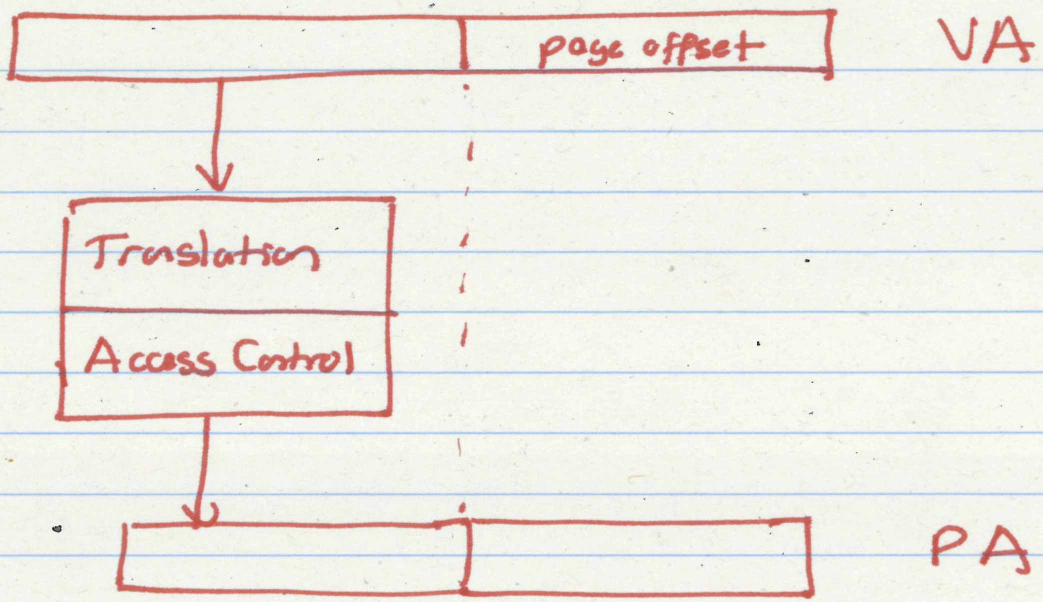
- Enforce access control at the same time as translation

→ Virtual memory system serves two functions today

Address translation (for illusion of large physical memory)

Access control (protection)

Two Functions of Virtual Memory



Virtual Memory

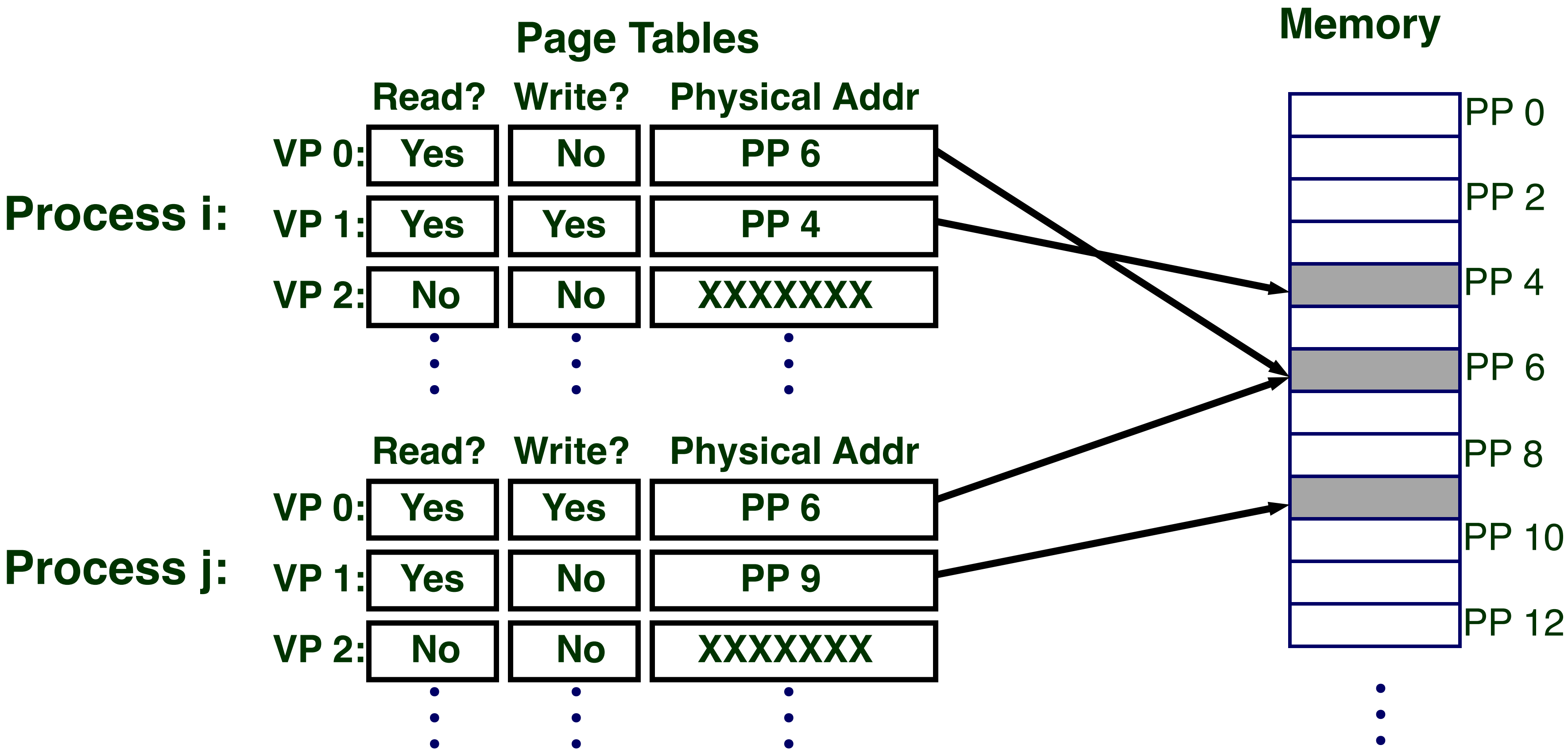
Two Functions Today

1. Translation
2. Access control (protection)

PTE contains access control bits associated with the virtual page.

VM as a Tool for Memory Access Protection

- Extend Page Table Entries (PTEs) with permission bits
- Check bits on each access and during a page fault
  - If violated, generate exception (Access Protection exception)



Page Tables

Process i:

	Read?	Write?	Physical Addr
VP 0:	Yes	No	PP 6
VP 1:	Yes	Yes	PP 4
VP 2:	No	No	XXXXXXX
	⋮	⋮	⋮

Process j:

	Read?	Write?	Physical Addr
VP 0:	Yes	Yes	PP 6
VP 1:	Yes	No	PP 9
VP 2:	No	No	XXXXXXX
	⋮	⋮	⋮

Privilege Levels in x86

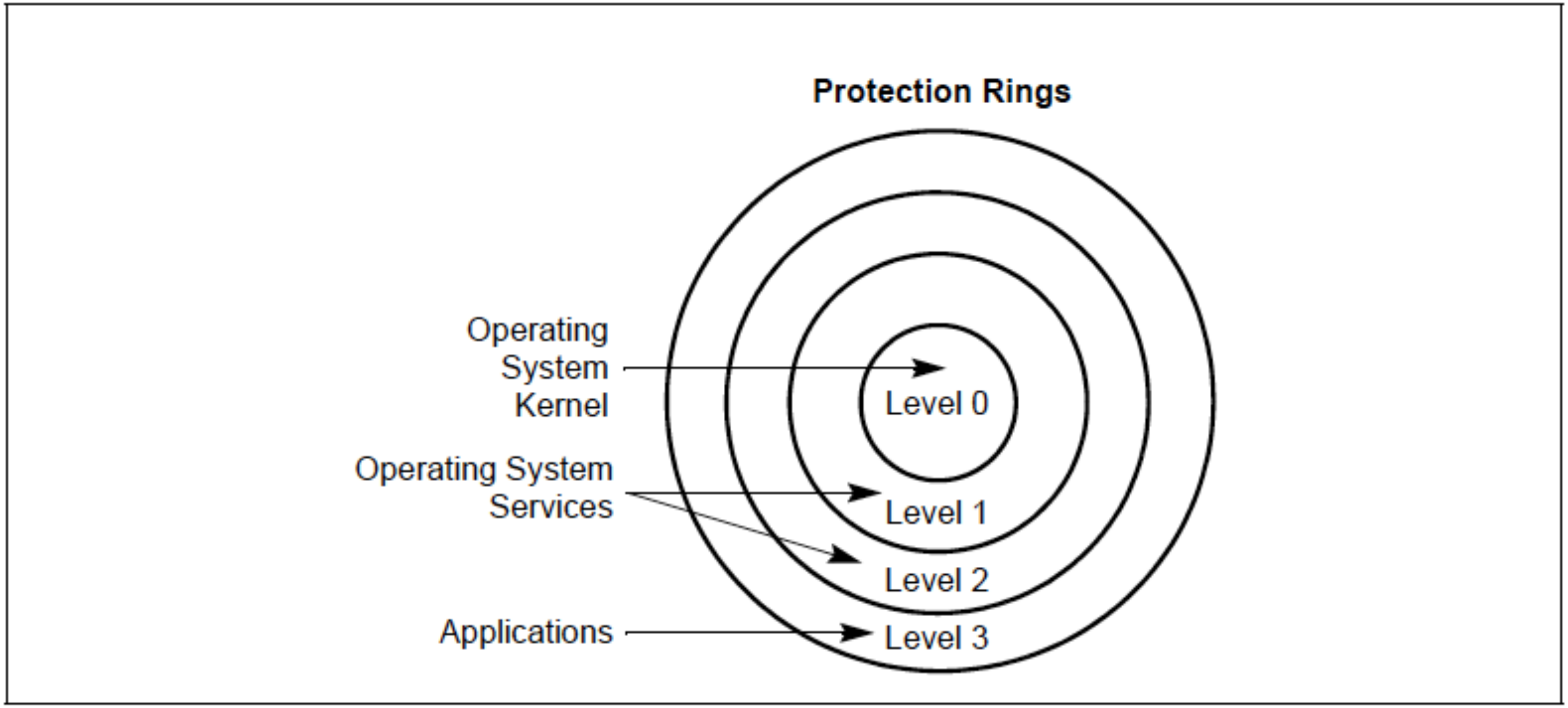


Figure 5-3. Protection Rings

Page Level Protection in x86

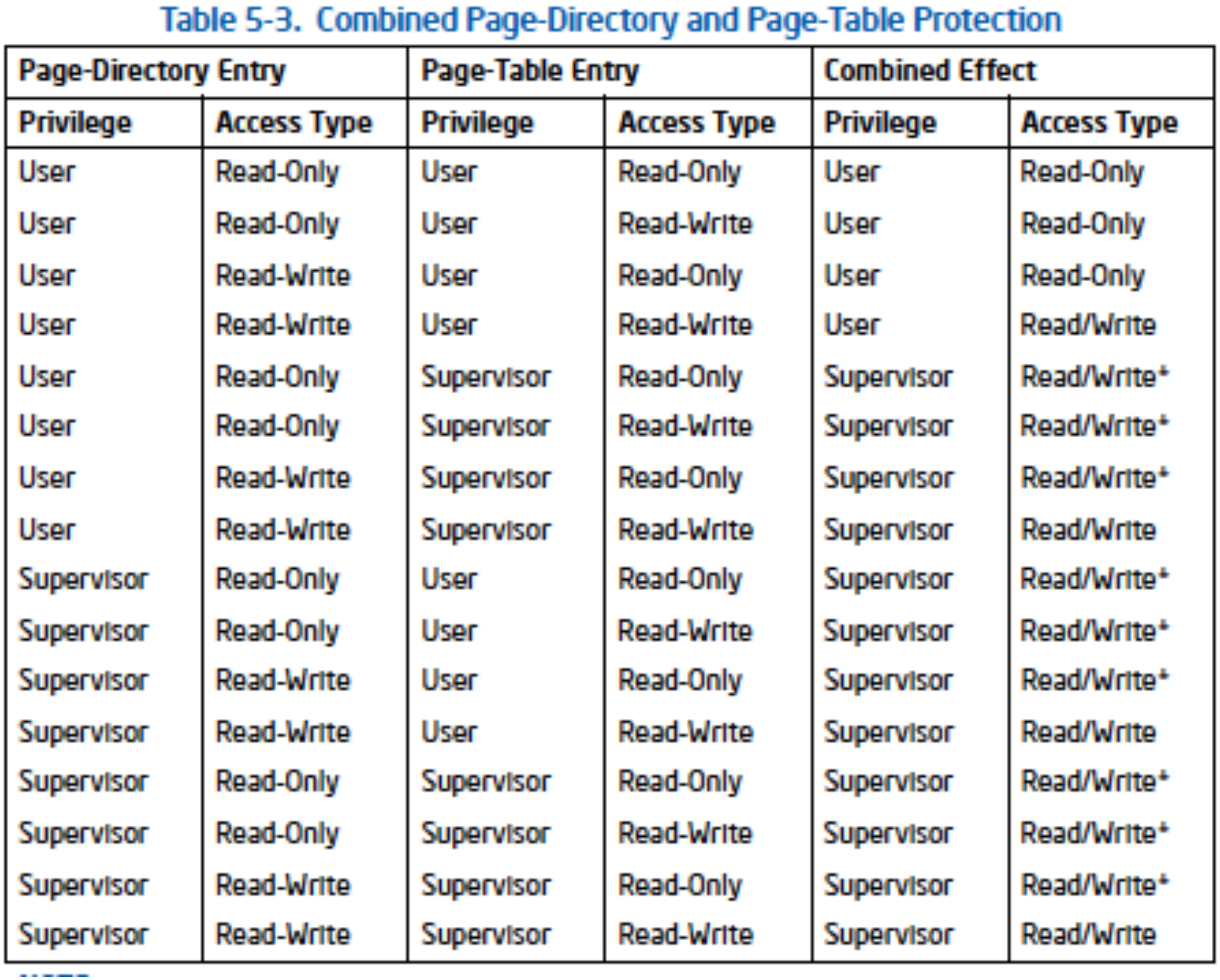


Table 5-3. Combined Page-Directory and Page-Table Protection

Page-Directory Entry		Page-Table Entry		Combined Effect	
Privilege	Access Type	Privilege	Access Type	Privilege	Access Type
User	Read-Only	User	Read-Only	User	Read-Only
User	Read-Only	User	Read-Write	User	Read-Only
User	Read-Write	User	Read-Only	User	Read-Only
User	Read-Write	User	Read-Write	User	Read/Write
User	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	User	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write

Food for Thought: What If?

- Your hardware is unreliable and someone can flip the access protection bits
  - such that a user-level program can gain supervisor-level access (i.e., access to all data on the system)
  - by flipping the access control bit from user to supervisor!

- Can this happen?

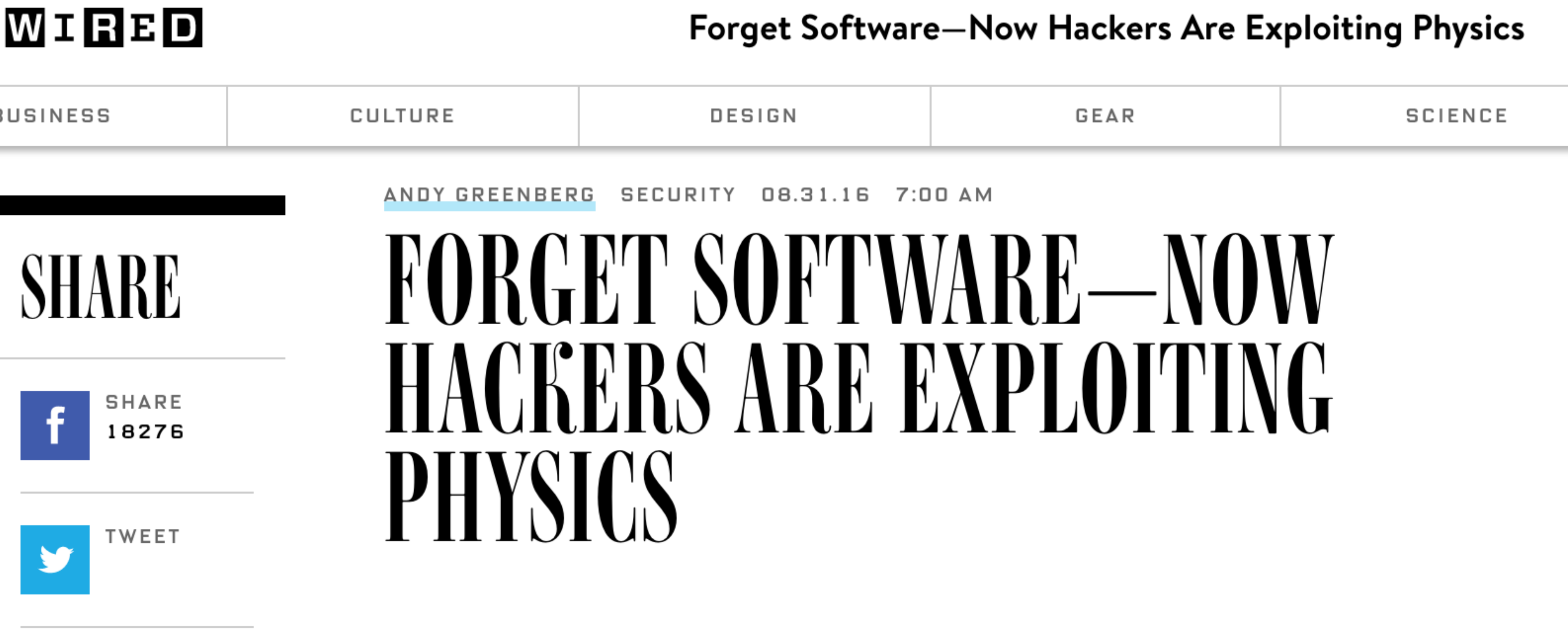
Remember RowHammer?

One can
predictably induce errors
in most DRAM memory chips

Remember RowHammer?

- DRAM Row Hammer (or, DRAM Disturbance Errors)

- How a simple hardware failure mechanism can create a widespread system security vulnerability



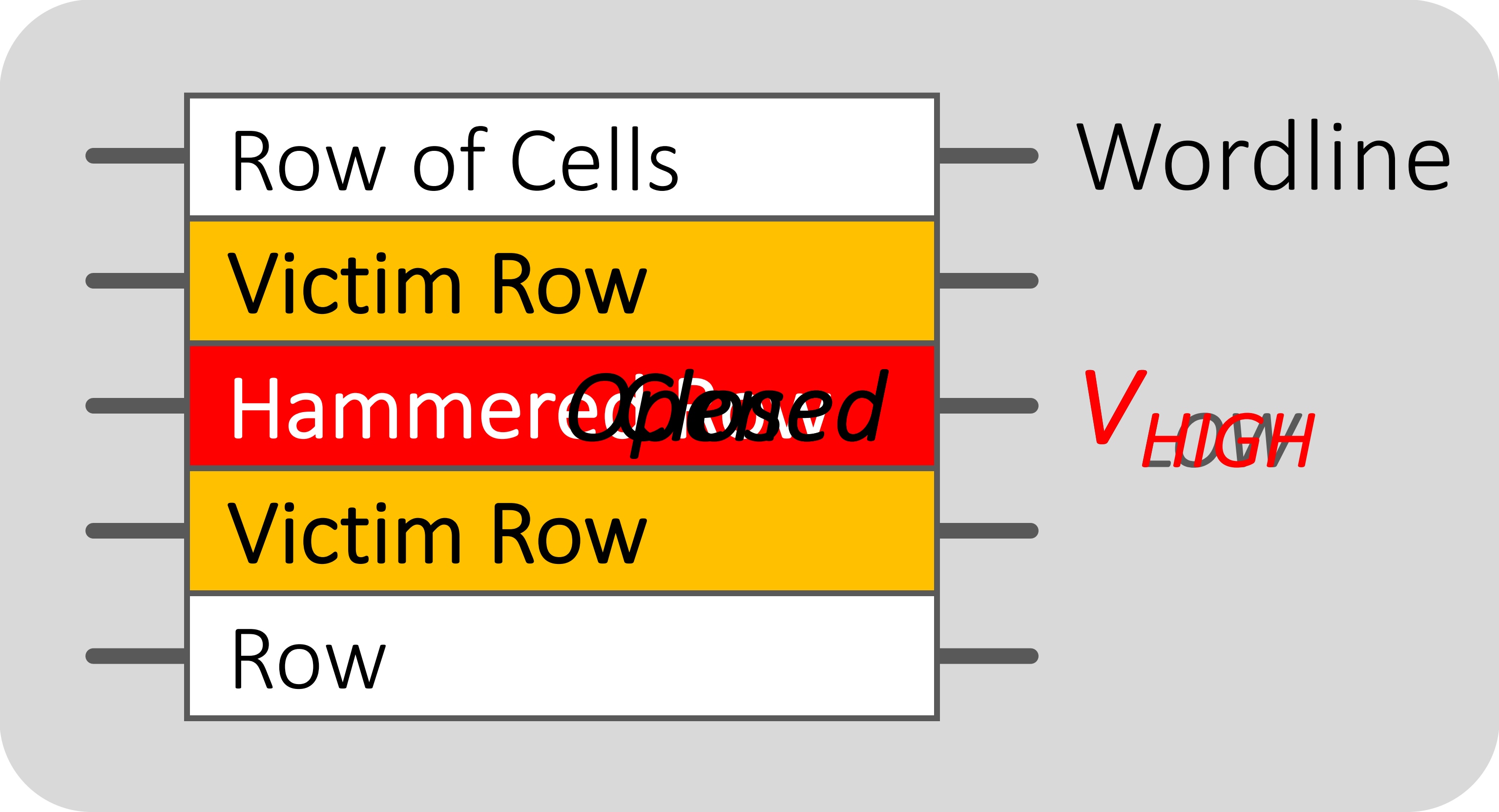
WIRED

Forget Software—Now Hackers Are Exploiting Physics

ANDY GREENBERG SECURITY 08.31.16 7:00 AM

FORGET SOFTWARE—NOW HACKERS ARE EXPLOITING PHYSICS

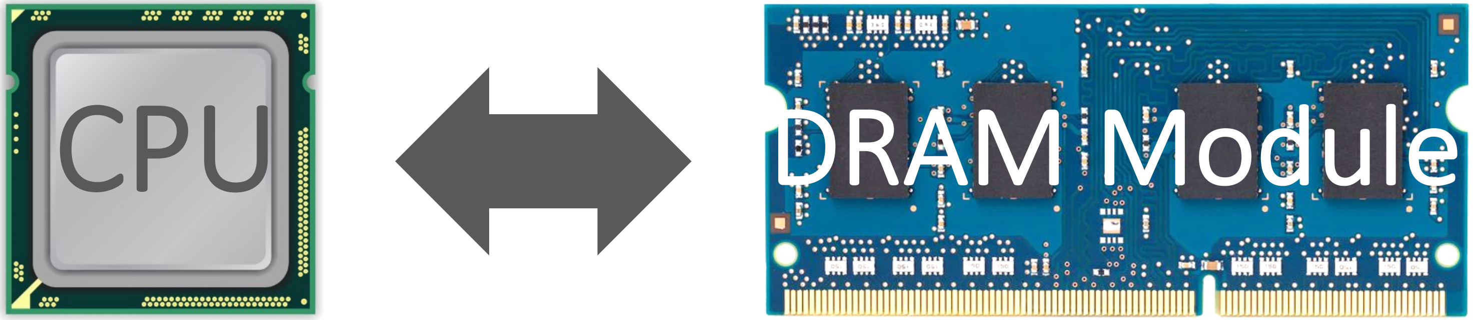
Modern DRAM is Prone to Disturbance Errors



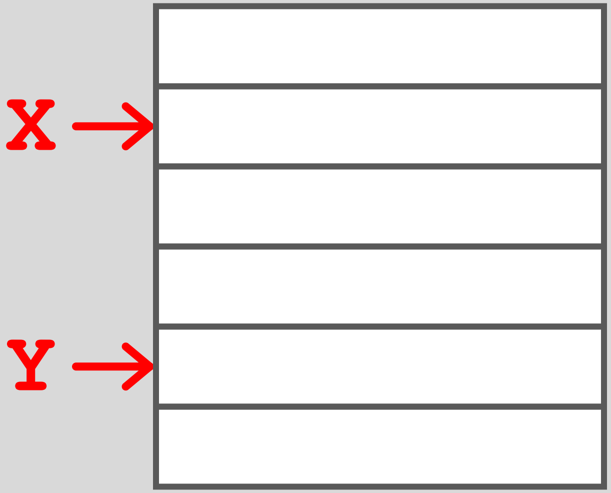
Repeatedly reading a row enough times (before memory gets refreshed) induces disturbance errors in **adjacent rows** in most real DRAM chips you can buy today

Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors, (Kim et al., ISCA 2014)

A Simple Program Can Induce Many Errors

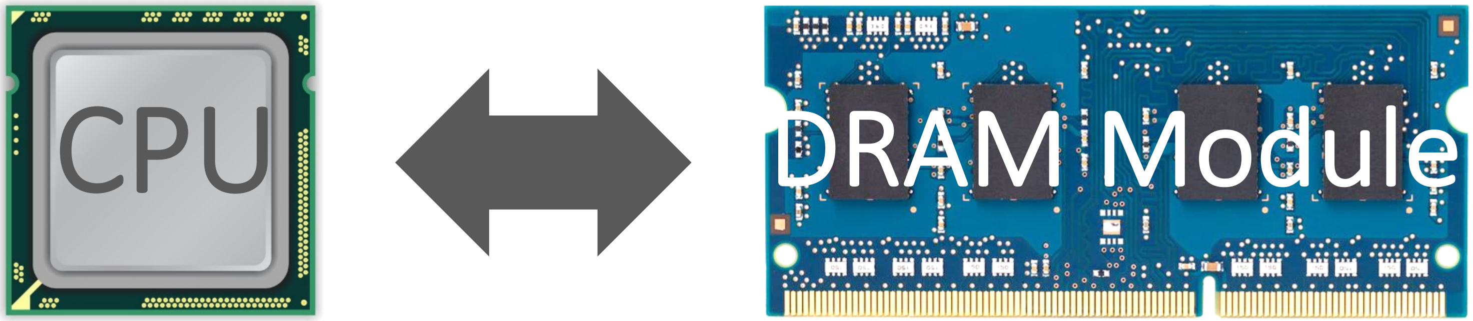


```
loop:
   mov  (X), %eax
   mov  (Y), %ebx
   clflush (X)
   clflush (Y)
   mfence
   jmp loop
```



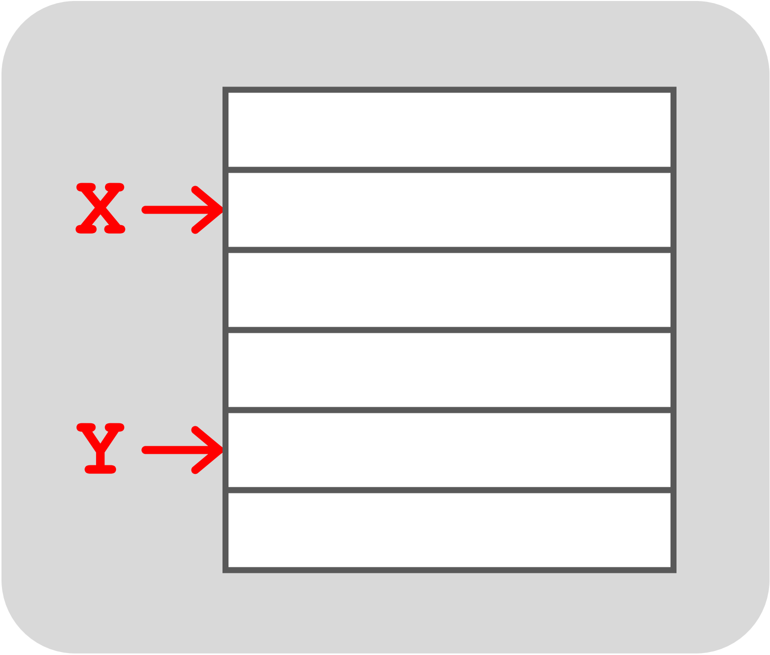
Download from: https://github.com/CMU-SAFARI/rowhammer

A Simple Program Can Induce Many Errors



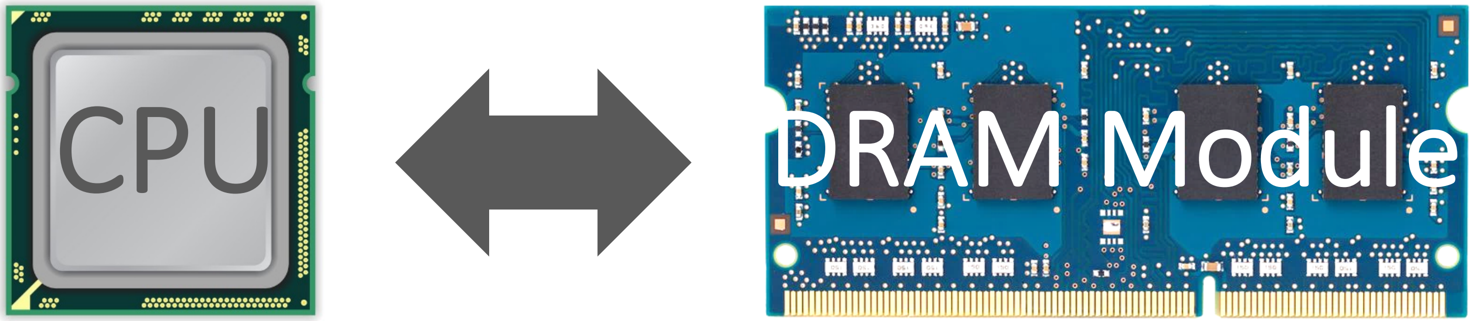
1. Avoid *cache hits*
   - Flush X from cache

2. Avoid *row hits* to X
   - Read Y in another row

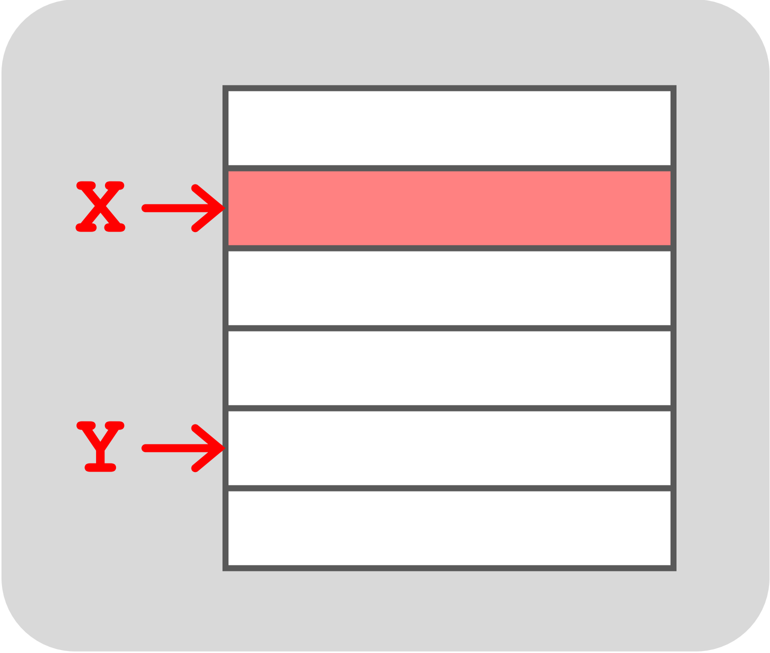


Download from: https://github.com/CMU-SAFARI/rowhammer

A Simple Program Can Induce Many Errors

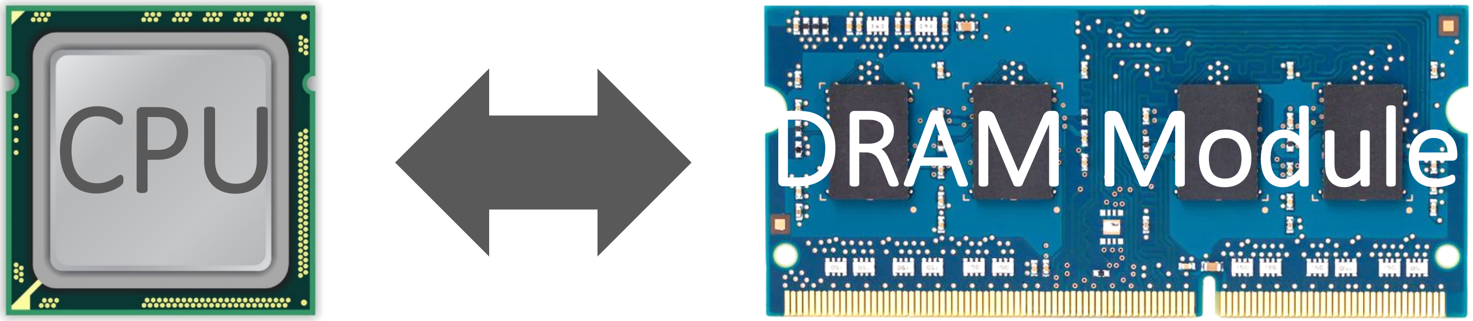


```
loop:
   mov  (X), %eax
   mov  (Y), %ebx
   clflush (X)
   clflush (Y)
   mfence
   jmp loop
```

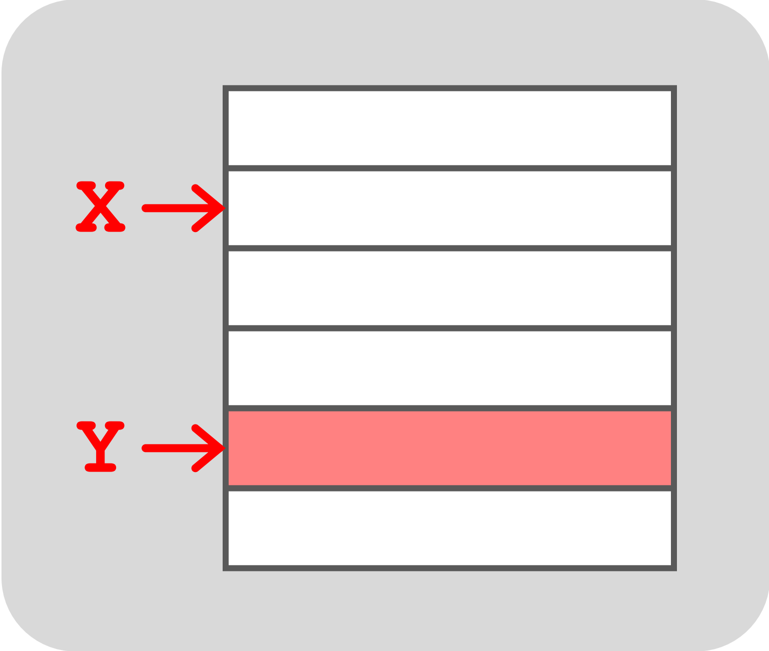


Download from: https://github.com/CMU-SAFARI/rowhammer

A Simple Program Can Induce Many Errors

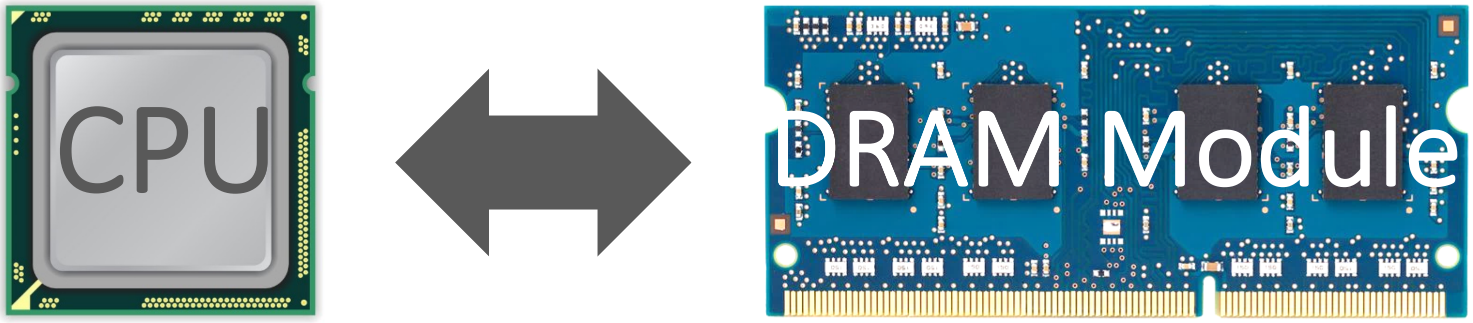


```
loop:
   mov  (X), %eax
   mov  (Y), %ebx
   clflush (X)
   clflush (Y)
   mfence
   jmp loop
```

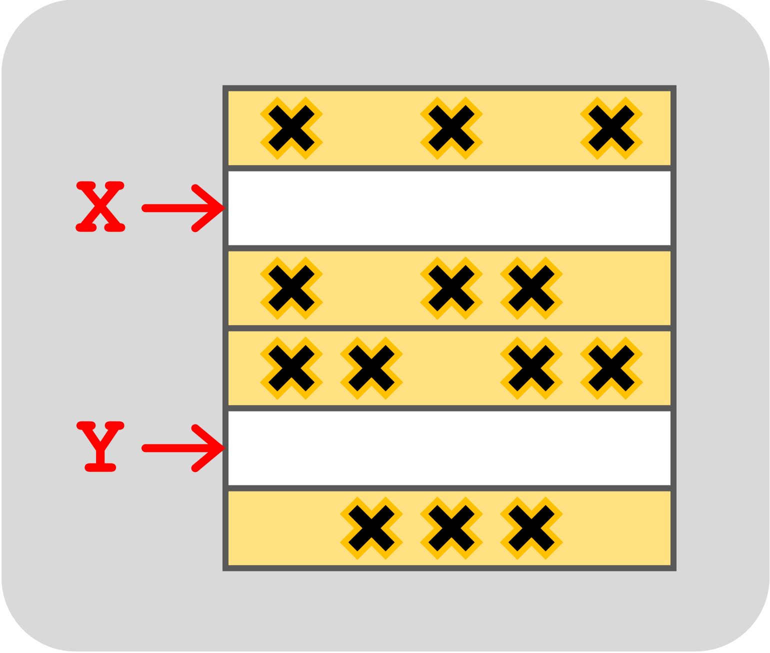


Download from: https://github.com/CMU-SAFARI/rowhammer

A Simple Program Can Induce Many Errors

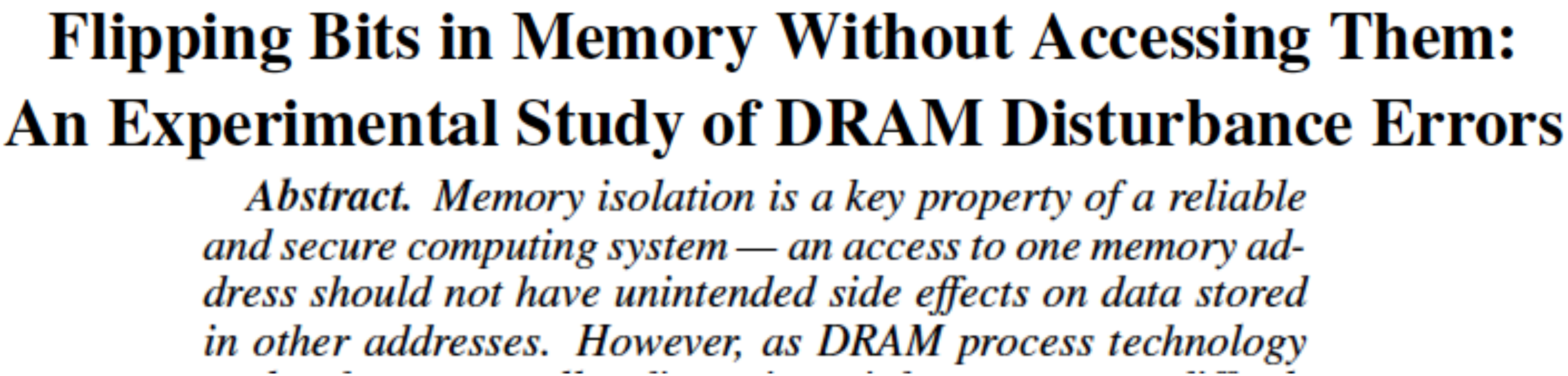


```
loop:
  mov (X), %eax
  mov (Y), %ebx
  clflush (X)
  clflush (Y)
  mfence
  jmp loop
```



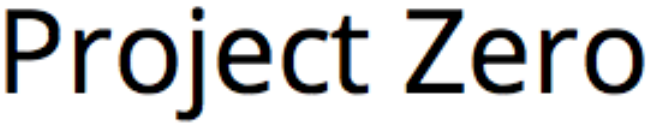
Download from: https://github.com/CMU-SAFARI/rowhammer

One Can Take Over an Otherwise-Secure System



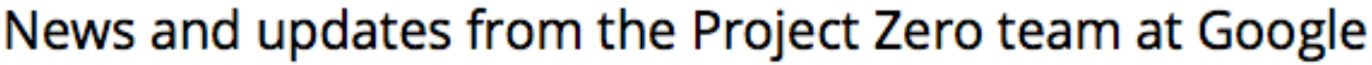
Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors

Abstract. *Memory isolation is a key property of a reliable and secure computing system — an access to one memory address should not have unintended side effects on data stored in other addresses. However, as DRAM process technology*



Project Zero

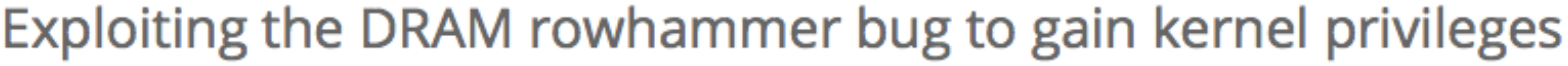
Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors (Kim et al., ISCA 2014)



News and updates from the Project Zero team at Google

Exploiting the DRAM rowhammer bug to gain kernel privileges (Seaborn, 2015)

Monday, March 9, 2015



Exploiting the DRAM rowhammer bug to gain kernel privileges

RowHammer Security Attack Example

- "Rowhammer" is a problem with some recent DRAM devices in which repeatedly accessing a row of memory can cause bit flips in adjacent rows (Kim et al., ISCA 2014).
  - Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors (Kim et al., ISCA 2014)

- We tested a selection of laptops and found that a subset of them exhibited the problem.
- We built two working privilege escalation exploits that use this effect.
  - Exploiting the DRAM rowhammer bug to gain kernel privileges (Seaborn+, 2015)

- One exploit uses rowhammer-induced bit flips to gain kernel privileges on x86-64 Linux when run as an unprivileged userland process.
- When run on a machine vulnerable to the rowhammer problem, the process was able to induce bit flips in page table entries (PTEs).
- It was able to use this to gain write access to its own page table, and hence gain read-write access to all of physical memory.

Exploiting the DRAM rowhammer bug to gain kernel privileges (Seaborn & Dullien, 2015)

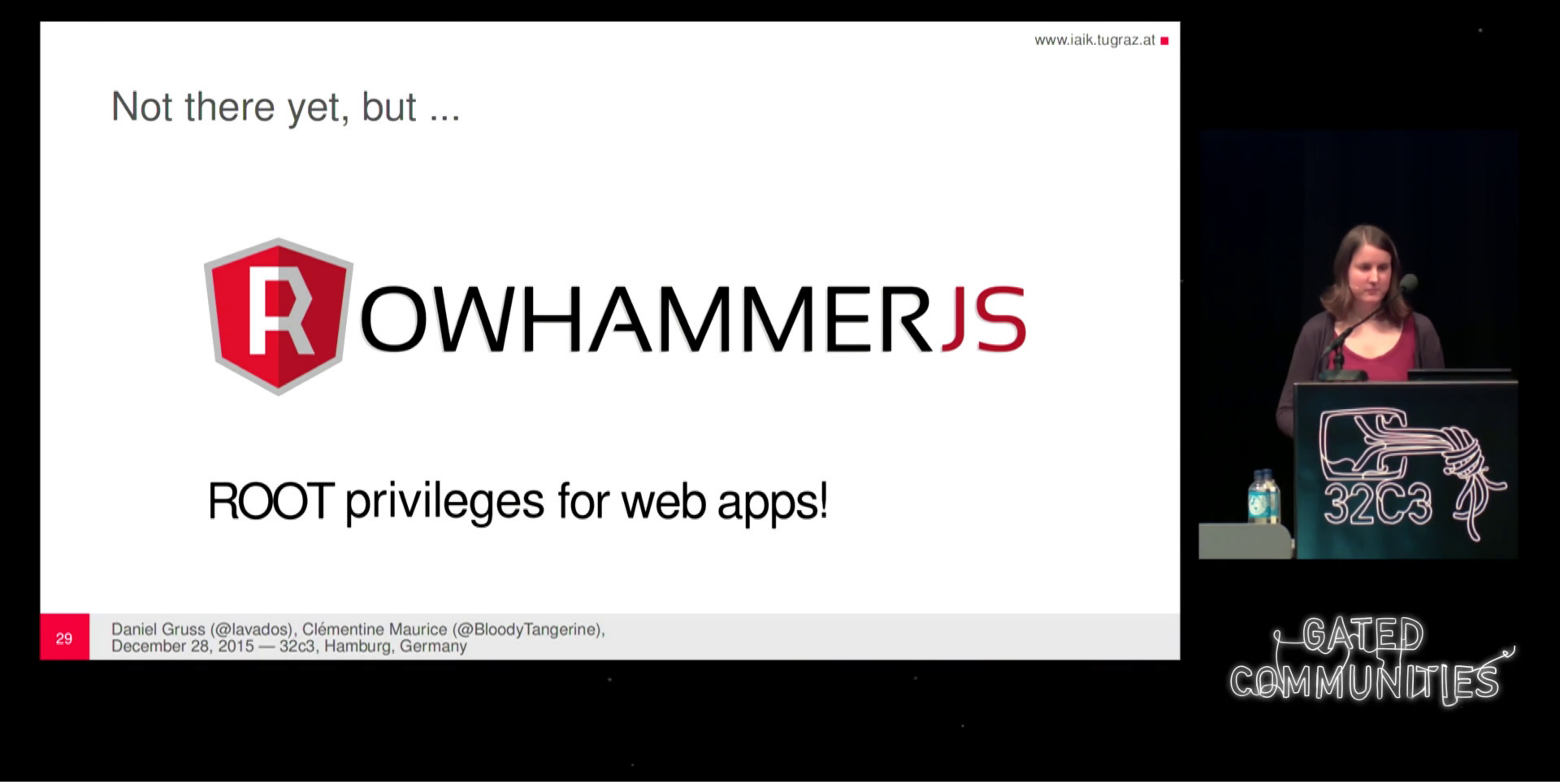
Security Implications

Security Implications

It’s like breaking into an apartment by repeatedly slamming a neighbor’s door until the vibrations open the door you were after

More Security Implications (I)

"We can gain unrestricted access to systems of website visitors."



www.iaik.tugraz.at

Not there yet, but ...

ROWHAMMERJS

ROOT privileges for web apps!

29

Daniel Gruss (@lavados), Clémentine Maurice (@BloodyTangerine),
December 28, 2015 — 32c3, Hamburg, Germany

Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript (DIMVA'16)

Source: https://lab.dsst.io/32c3-slides/7197.html

More Security Implications (II)

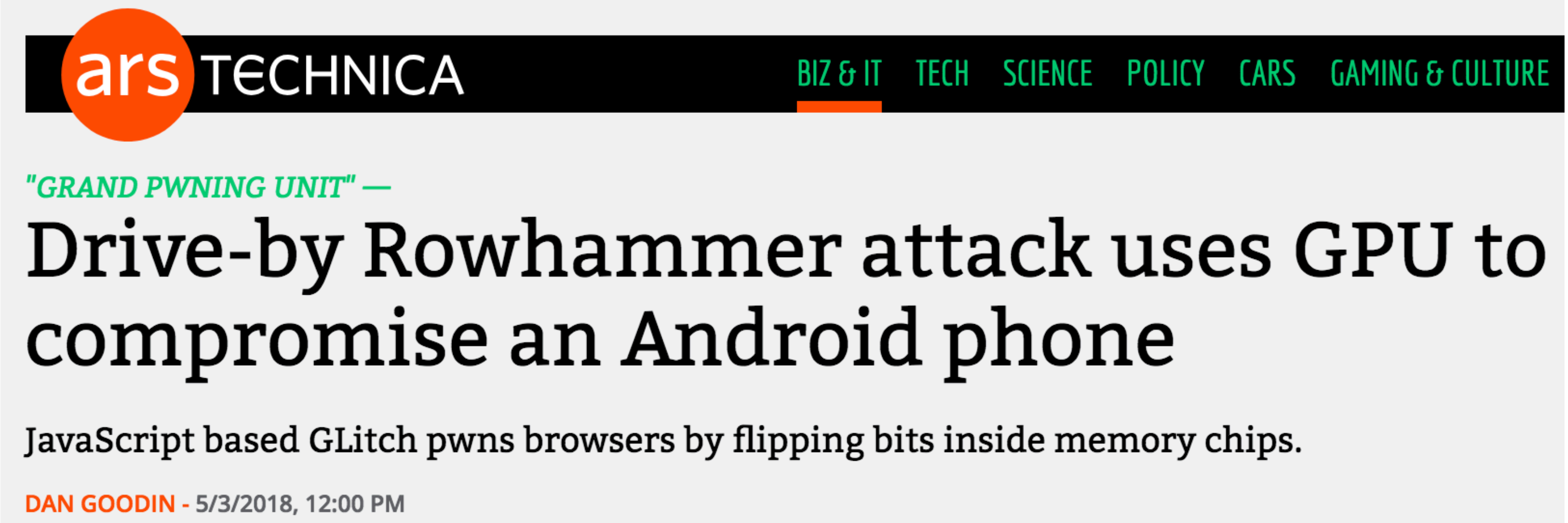
"Can gain control of a smart phone deterministically"

Drammer: Deterministic Rowhammer Attacks on Mobile Platforms, CCS'16

Source: https://fossbytes.com/drammer-rowhammer-attack-android-root-devices/

More Security Implications (III)

- Using an integrated GPU in a mobile system to remotely escalate privilege via the WebGL interface

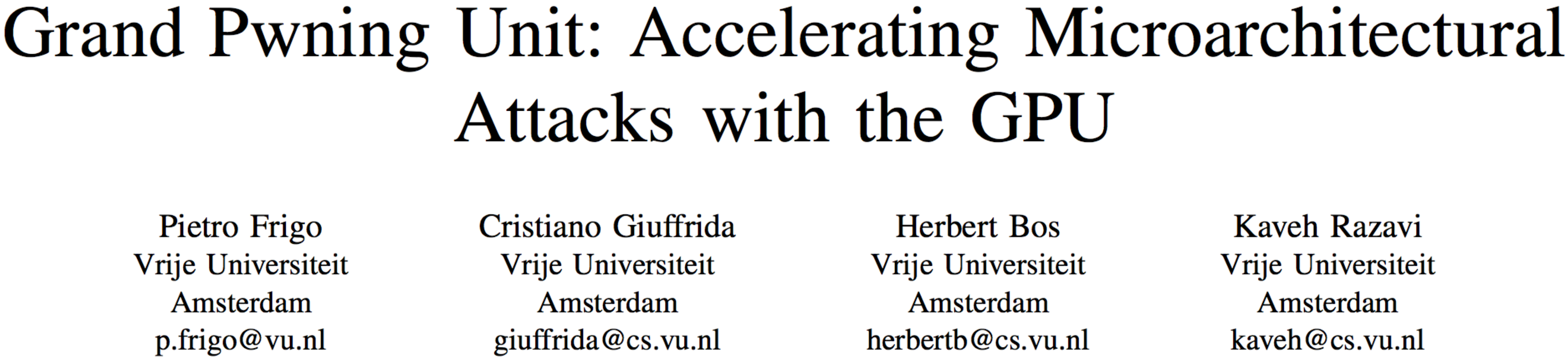


"GRAND PWNING UNIT" —

Drive-by Rowhammer attack uses GPU to compromise an Android phone

JavaScript based GLitch pwns browsers by flipping bits inside memory chips.

DAN GOODIN - 5/3/2018, 12:00 PM



Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU

Pietro Frigo
Vrije Universiteit
Amsterdam
p.frigo@vu.nl

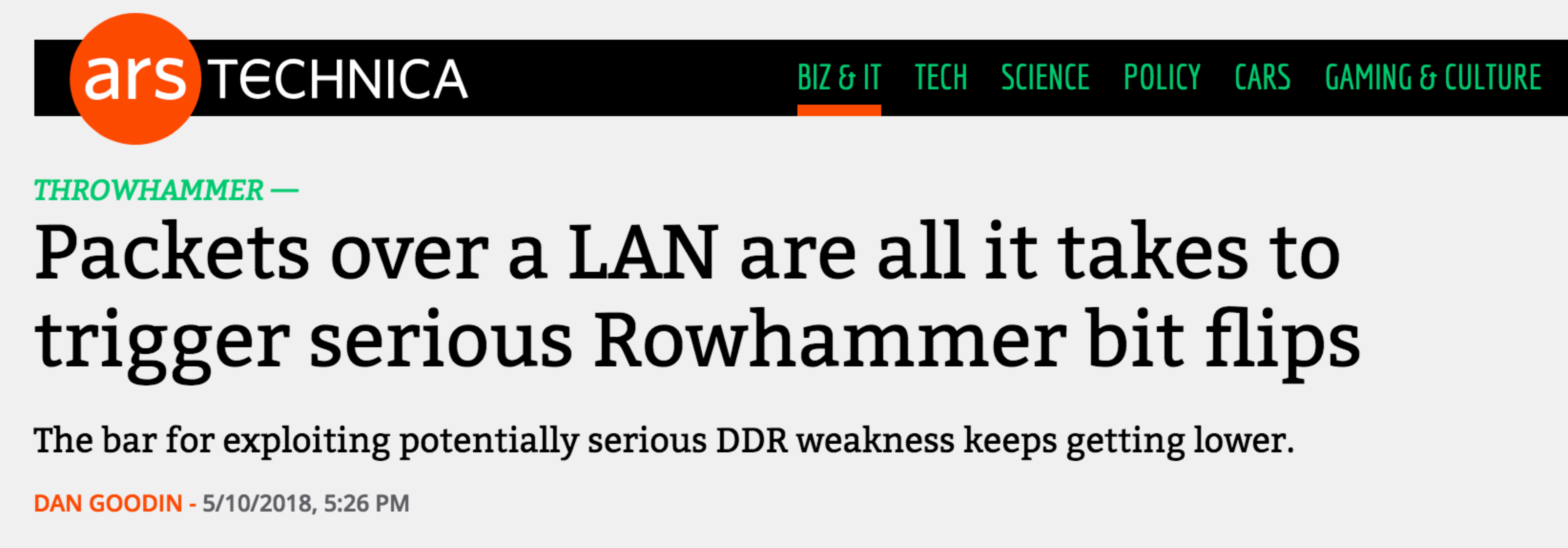
Cristiano Giuffrida
Vrije Universiteit
Amsterdam
giuffrida@cs.vu.nl

Herbert Bos
Vrije Universiteit
Amsterdam
herbertb@cs.vu.nl

Kaveh Razavi
Vrije Universiteit
Amsterdam
kaveh@cs.vu.nl

More Security Implications (IV)

- Rowhammer over RDMA (I)

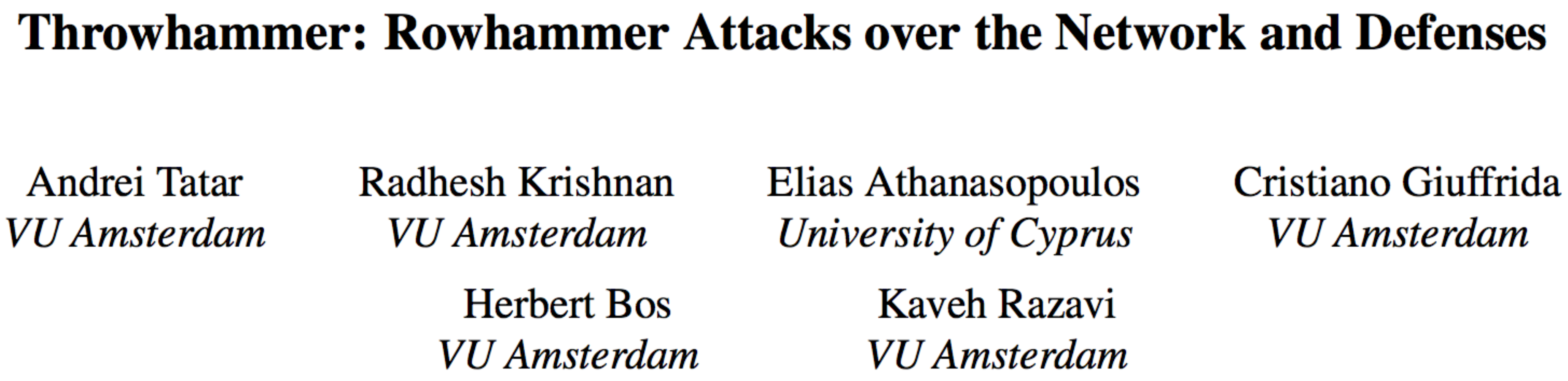


THROWHAMMER —

Packets over a LAN are all it takes to trigger serious Rowhammer bit flips

The bar for exploiting potentially serious DDR weakness keeps getting lower.

DAN GOODIN - 5/10/2018, 5:26 PM



Throwhammer: Rowhammer Attacks over the Network and Defenses

Andrei Tatar
VU Amsterdam

Radhesh Krishnan
VU Amsterdam

Elias Athanasopoulos
University of Cyprus

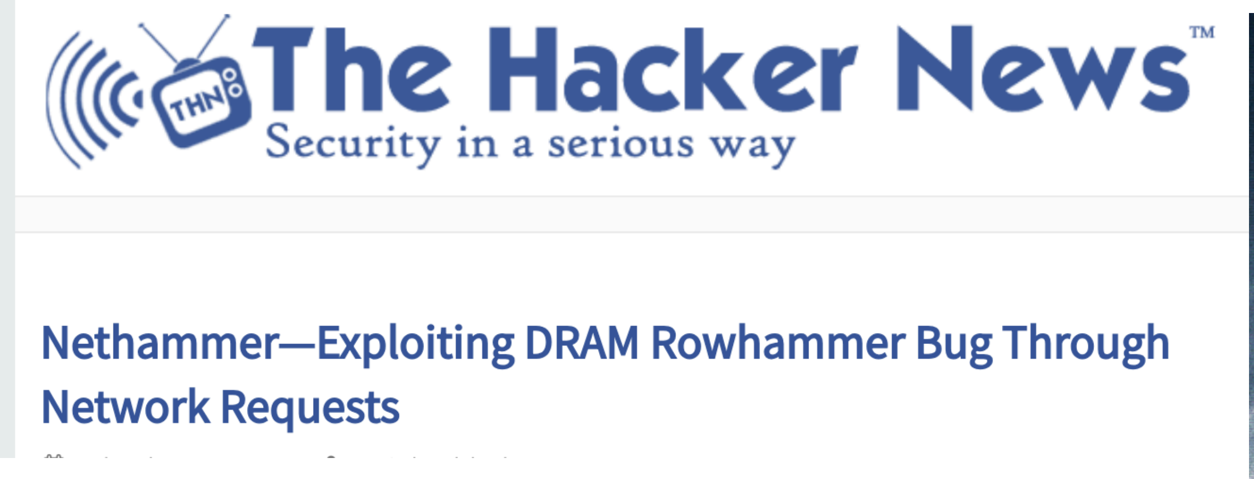
Cristiano Giuffrida
VU Amsterdam

Herbert Bos
VU Amsterdam

Kaveh Razavi
VU Amsterdam

More Security Implications (V)

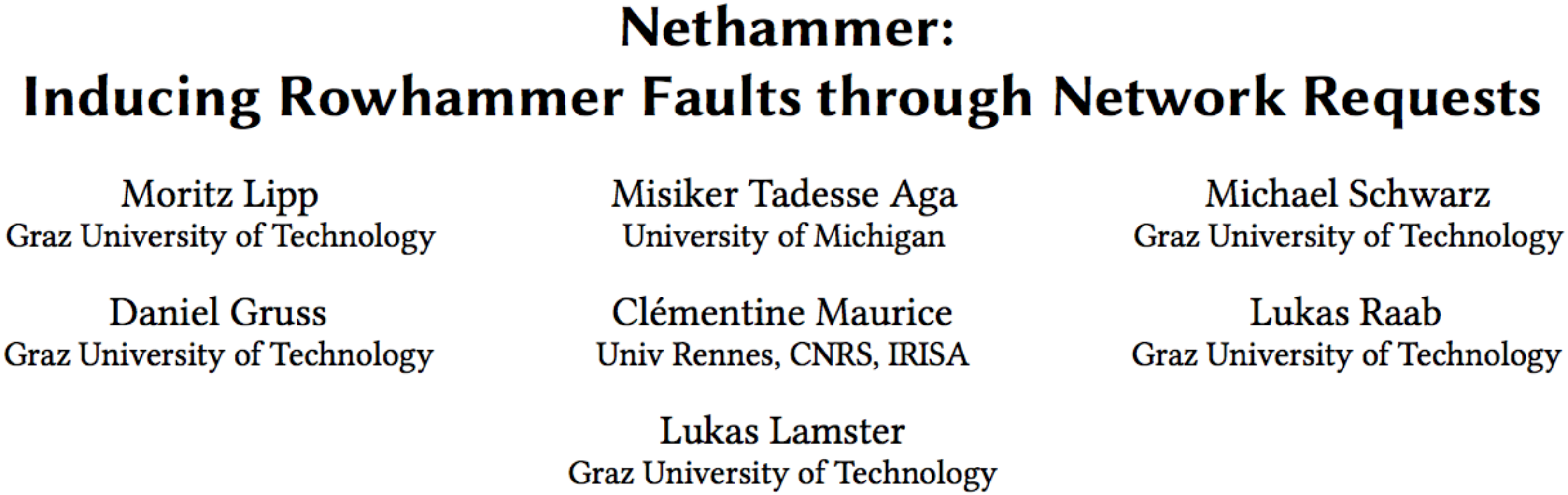
- Rowhammer over RDMA (II)



The Hacker News™

Security in a serious way

Nethammer—Exploiting DRAM Rowhammer Bug Through Network Requests



Nethammer: Inducing Rowhammer Faults through Network Requests

Moritz Lipp
Graz University of Technology

Misiker Tadesse Aga
University of Michigan

Michael Schwarz
Graz University of Technology

Daniel Gruss
Graz University of Technology

Clémentine Maurice
Univ Rennes, CNRS, IRISA

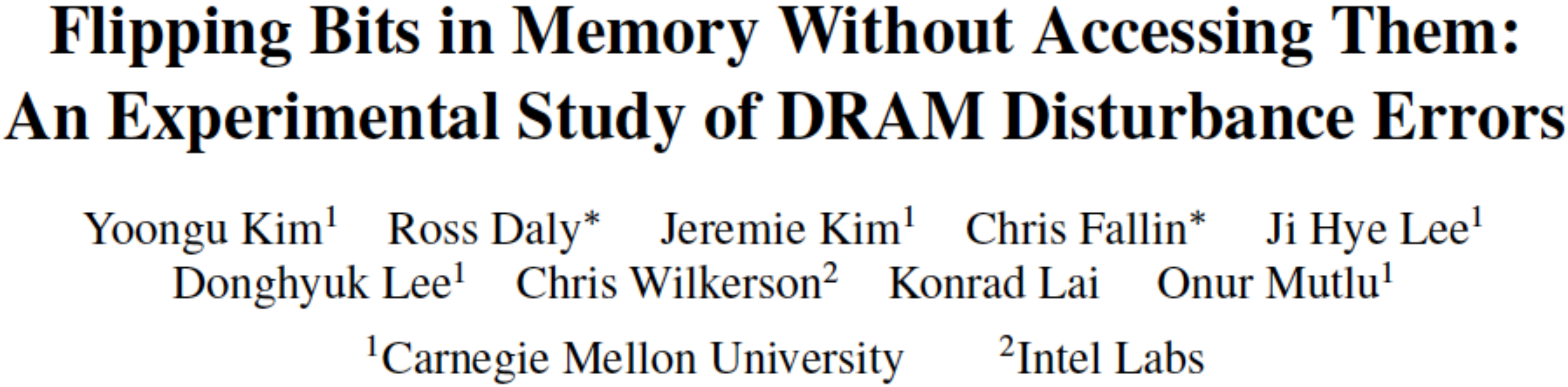
Lukas Raab
Graz University of Technology

Lukas Lamster
Graz University of Technology

More Security Implications?

Curious? First RowHammer Paper

- Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu,
"Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors"
Proceedings of the 41st International Symposium on Computer Architecture (***ISCA***), Minneapolis, MN, June 2014.
[Slides (pptx) (pdf)] [Lightning Session Slides (pptx) (pdf)] [Source Code and Data]



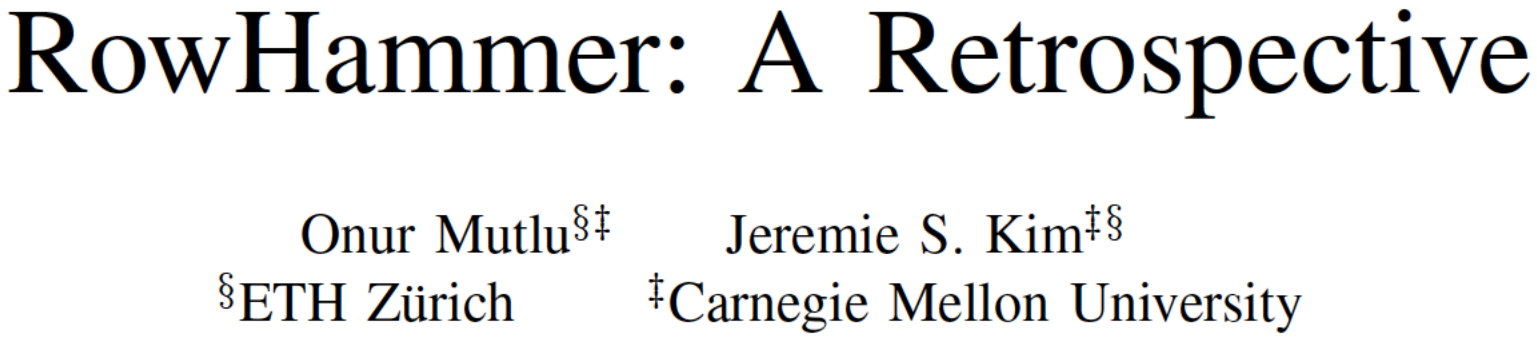
Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors

Yoongu Kim[1] Ross Daly* Jeremie Kim[1] Chris Fallin* Ji Hye Lee[1]
Donghyuk Lee[1] Chris Wilkerson[2] Konrad Lai Onur Mutlu[1]

[1]Carnegie Mellon University [2]Intel Labs

Curious? A RowHammer Retrospective

- Onur Mutlu and Jeremie Kim,
  "RowHammer: A Retrospective"
  IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (***TCAD***) *Special Issue on Top Picks in Hardware and Embedded Security*, 2019.
  [Preliminary arXiv version]



RowHammer: A Retrospective

Onur Mutlu[§‡] Jeremie S. Kim[‡§]

[§]ETH Zürich [‡]Carnegie Mellon University

Takeaway and Food for Thought

- If hardware is unreliable, higher-level security and protection mechanisms (as in virtual memory) may be compromised

- The root of security and trust is at the very low levels…
  - in the hardware itself
  - RowHammer, Spectre, Meltdown are recent key examples…

- What should we assume the hardware provides?
- How do we keep hardware reliable?
- How do we design secure hardware?
- How do we design secure hardware with high performance, high energy efficiency, low cost, convenient programming?

Plenty of exciting and highly-relevant research questions

Some Issues in Virtual Memory

Three Major Issues

1. How large is the page table and how do we store and access it?

2. How can we speed up translation & access control check?

3. When do we do the translation in relation to cache access?

- There are many other issues we will not cover in detail
  - What happens on a context switch?
  - How can you handle multiple page sizes?
  - …

Teaser: Virtual Memory Issue III

- When do we do the address translation?
  - Before or after accessing the L1 cache?

Address Translation and Caching

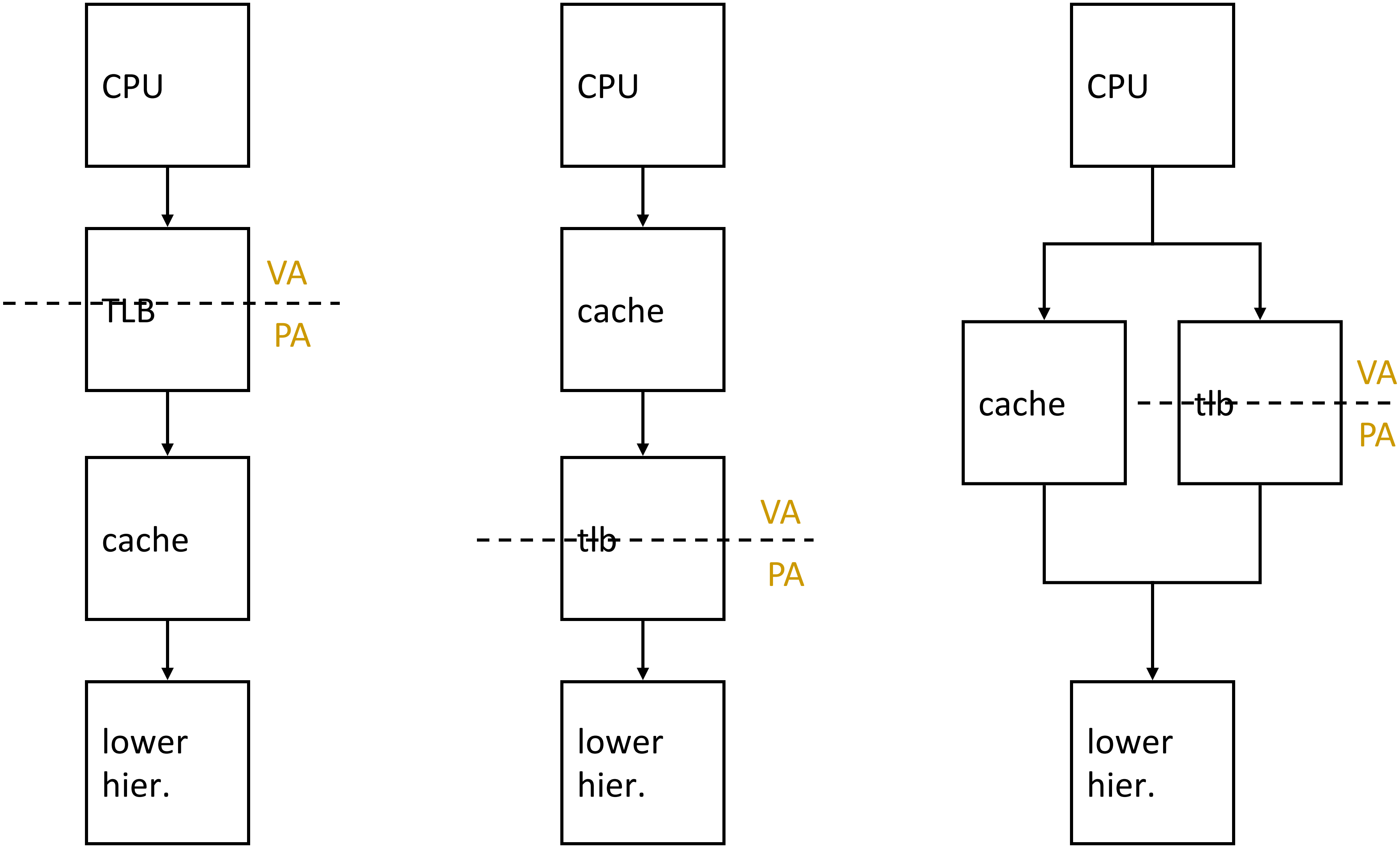
- When do we do the address translation?
  - Before or after accessing the L1 cache?

- In other words, is the cache virtually addressed or physically addressed?
  - Virtual versus physical cache

- What are the issues with a virtually addressed cache?

- Synonym problem:
  - Two different virtual addresses can map to the same physical address → same physical address can be present in multiple locations in the cache → can lead to inconsistency in data

Cache-VM Interaction



physical cache

virtual (L1) cache

virtual-physical cache

Virtual Memory
Summary

Virtual Memory Summary

- Virtual memory gives the illusion of "infinite" capacity
- A subset of virtual pages are located in physical memory
- A page table maps virtual pages to physical pages – this is called address translation
- A TLB speeds up address translation
- Multi-level page tables keep the page table size in check
- Using different page tables for different programs provides memory protection

Virtual Memory: Parting Thoughts

- VM is one of the most successful examples of
  - architectural support for programmers
  - how to partition work between hardware and software
  - hardware/software cooperation
  - programmer/architect tradeoff

- Going forward: How does virtual memory scale into the future? Three key trends:
  - Increasing, huge physical memory sizes
  - Hybrid physical memory systems (DRAM + NVM + …)
  - Many accelerators in the system addressing physical memory

Design of Digital Circuits

Lecture 25: Virtual Memory II

Prof. Onur Mutlu

ETH Zurich

Spring 2019

24 May 2019

Some Issues in Virtual Memory

Three Major Issues

1. How large is the page table and how do we store and access it?

2. How can we speed up translation & access control check?

3. When do we do the translation in relation to cache access?

- There are many other issues we will not cover in detail
  - What happens on a context switch?
  - How can you handle multiple page sizes?
  - …

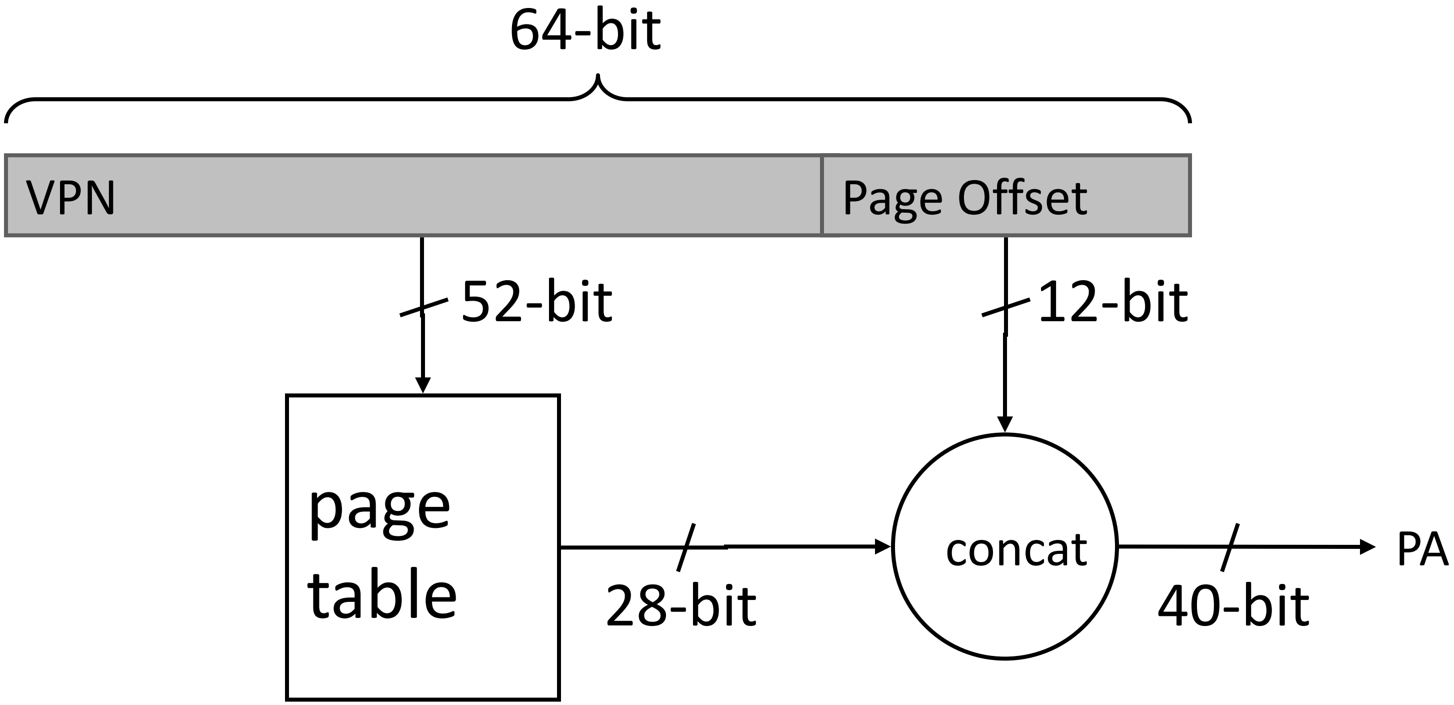
Virtual Memory Issue I

- How large is the page table?

- Where do we store it?
  - In hardware?
  - In physical memory? (Where is the PTBR?)
  - In virtual memory? (Where is the PTBR?)

- How can we store it efficiently without requiring physical memory that can store all page tables?
  - Idea: multi-level page tables
  - Only the first-level page table has to be in physical memory
  - Remaining levels are in virtual memory (but get cached in physical memory when accessed)

Issue: Page Table Size



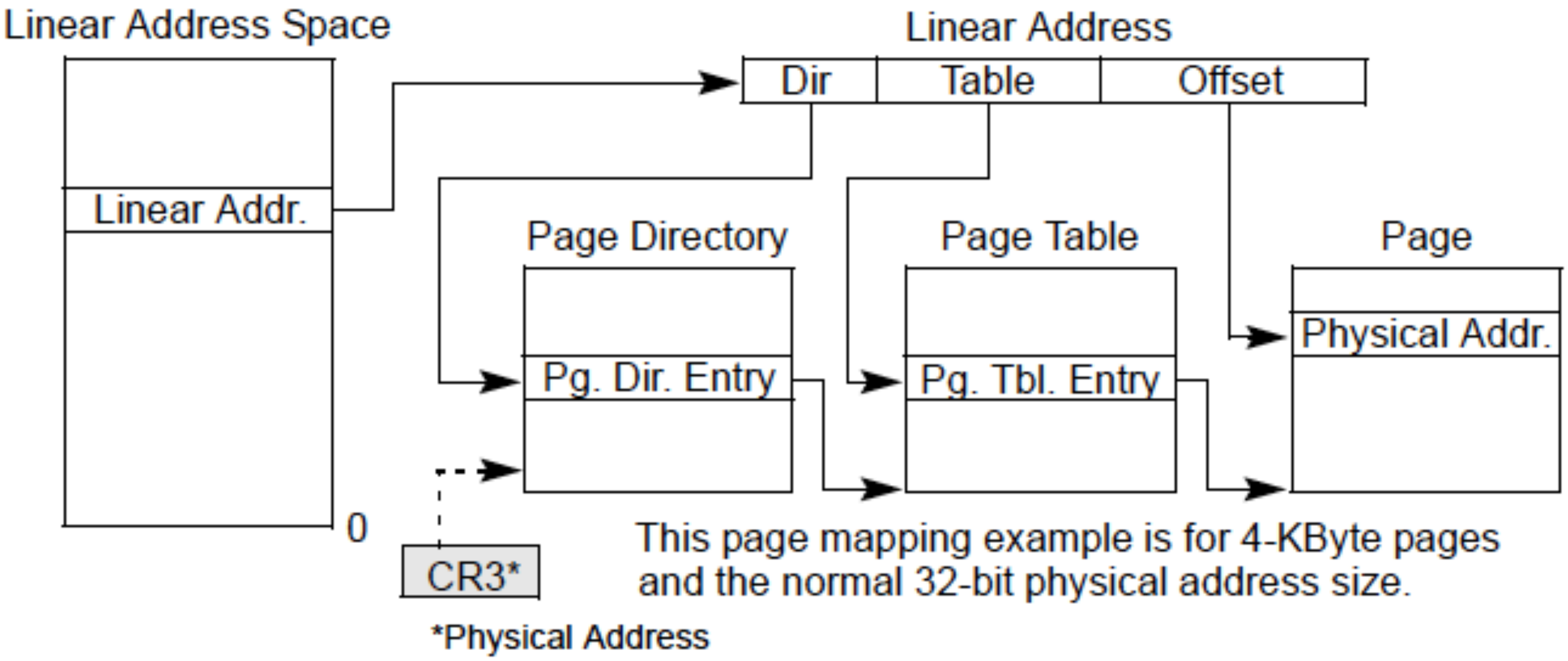
- Suppose 64-bit VA and 40-bit PA, how large is the page table?
  - **2^{52} entries x ~4 bytes ≈ 2^{54} bytes**

  and that is for just one process!

  and the process may not be using the entire VM space!

Solution: Multi-Level Page Tables

Example from the x86 architecture



Page Table Access

- How do we access the Page Table?

- Page Table Base Register (CR3 in x86)
- Page Table Limit Register

- If VPN is out of the bounds (exceeds PTLR) then the process did not allocate the virtual page → access control exception

- Page Table Base Register is part of a process's context
  - Just like PC, status registers, general purpose registers
  - Needs to be loaded when the process is context-switched in

More on x86 Page Tables (I): Small Pages

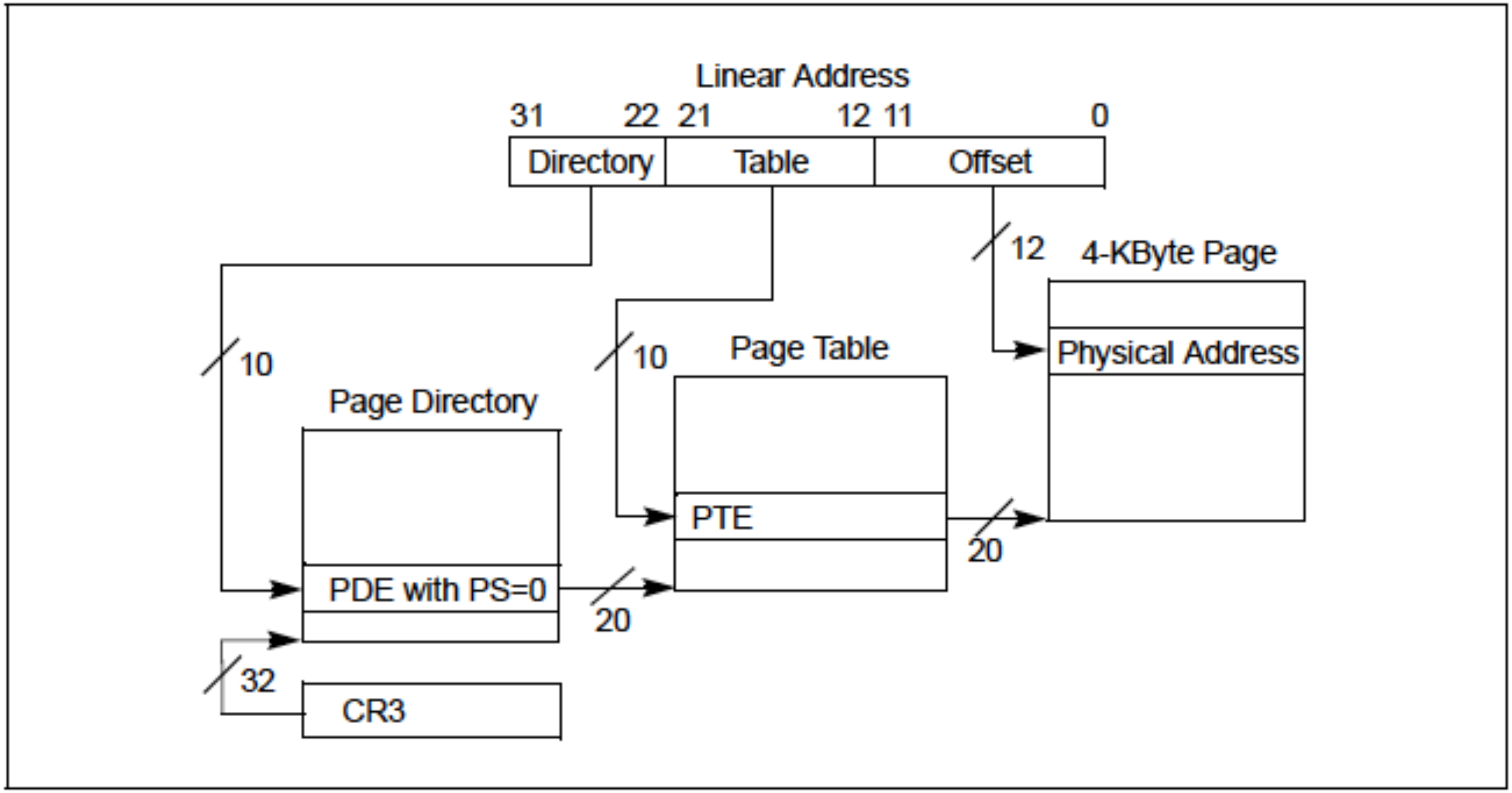


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

More on x86 Page Tables (II): Large Pages

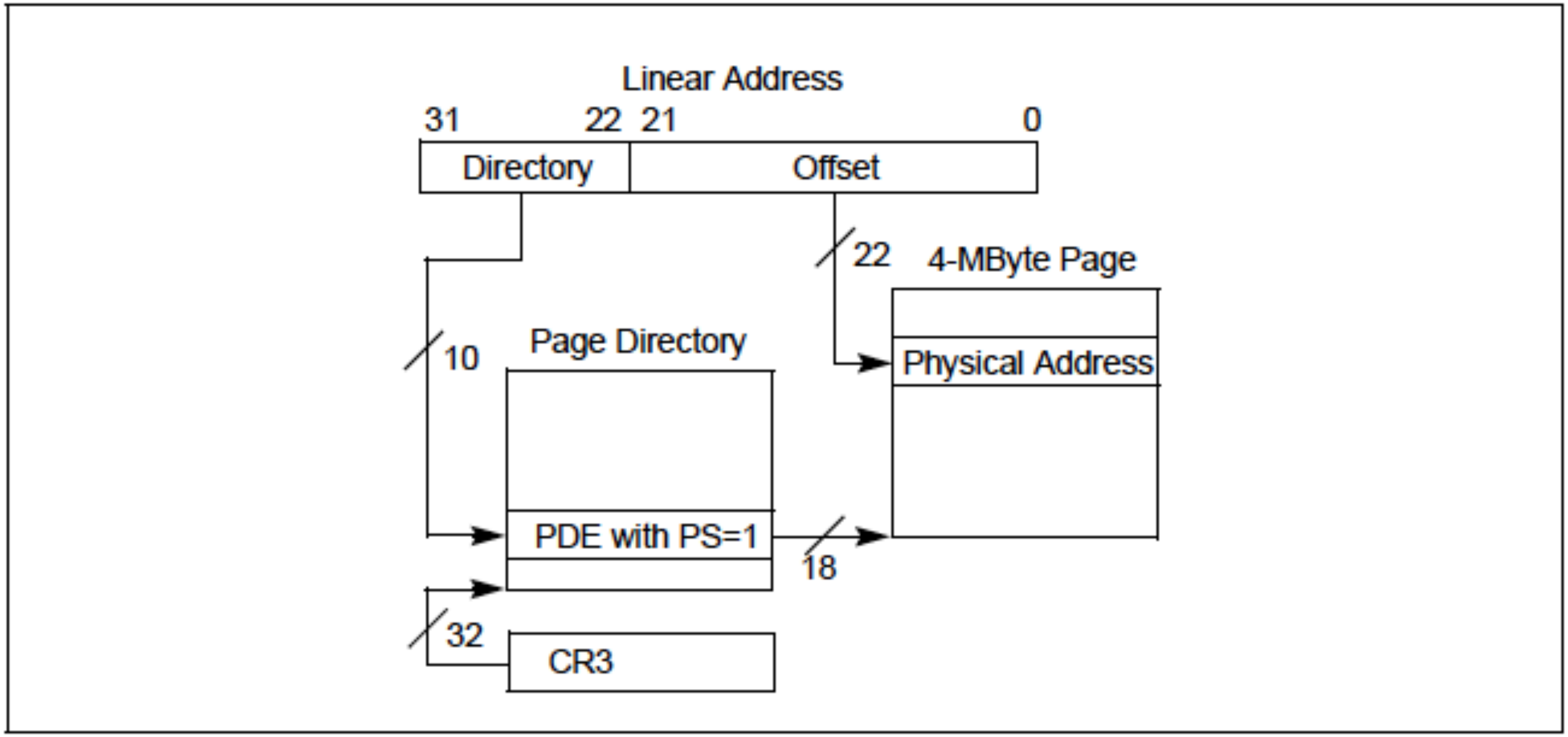
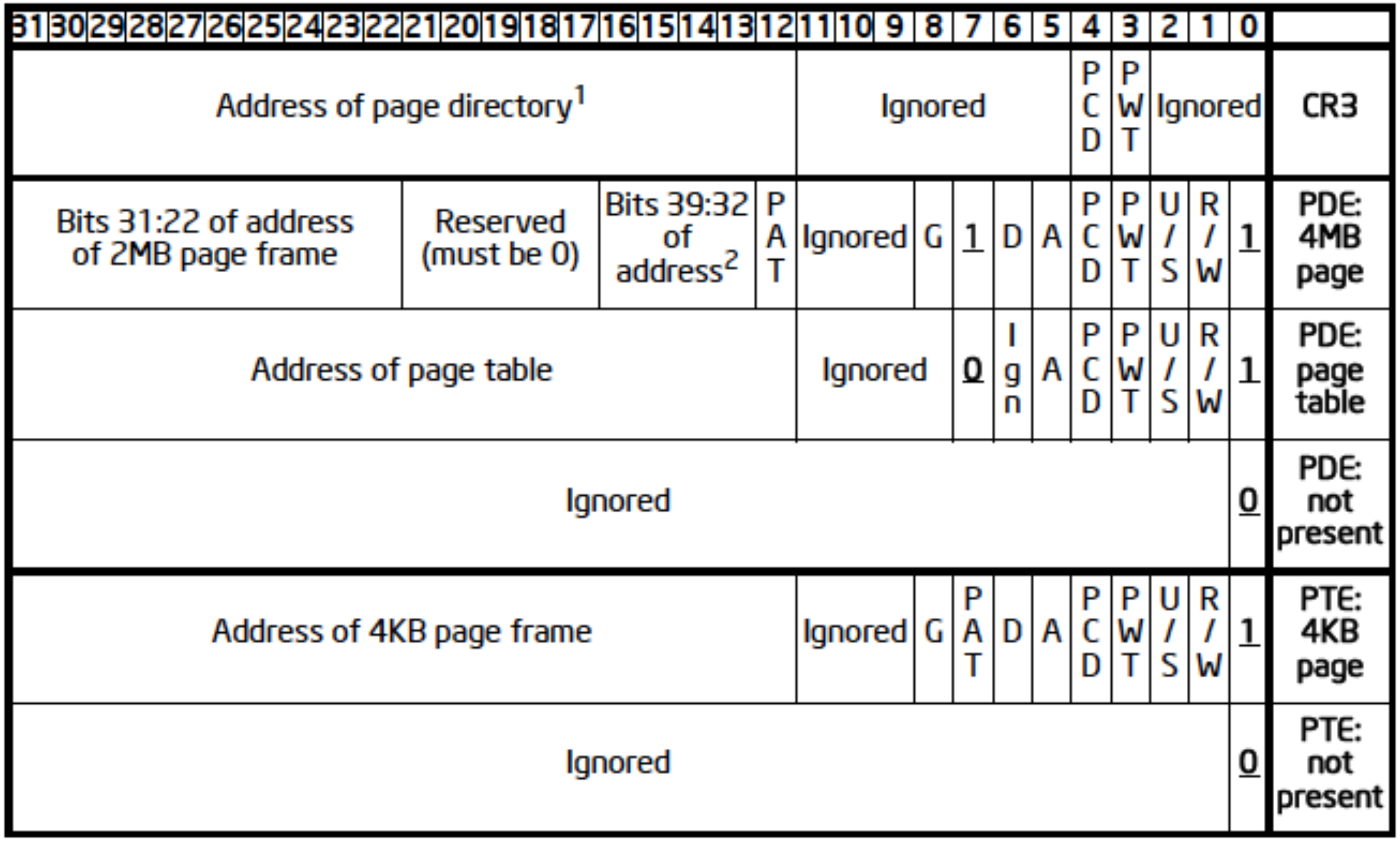


Figure 4-3. Linear-Address Translation to a 4-MByte Page using 32-Bit Paging

x86 Page Table Entries

Figure 4-4 gives a summary of the formats of CR3 and the paging-structure entries with 32-bit paging. For the paging structure entries, it identifies separately the format of entries that map pages, those that reference other paging structures, and those that do neither because they are "not present"; bit 0 (P) and bit 7 (PS) are highlighted because they determine how such an entry is used.



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]																				Ignored							PCD	PWT	Ignored			CR3
Bits 31:22 of address of 2MB page frame										Reserved (must be 0)					Bits 39:32 of address[2]				PAT	Ignored			G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																															0	PDE: not present
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																															0	PTE: not present

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

x86 PTE (4KB page)

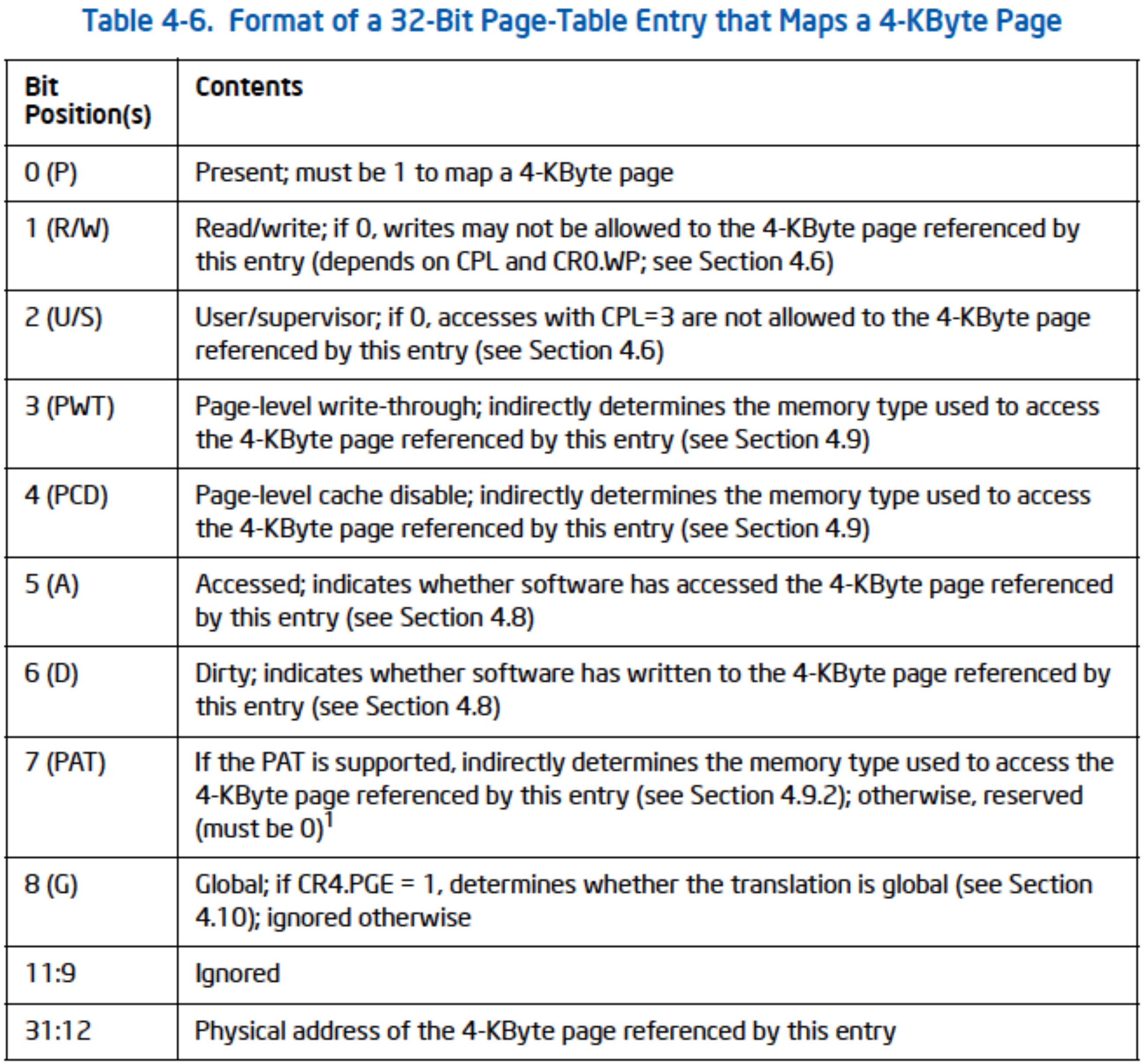


Table 4-6. Format of a 32-Bit Page-Table Entry that Maps a 4-KByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (depends on CPL and CR0.WP; see Section 4.6)
2 (U/S)	User/supervisor; if 0, accesses with CPL=3 are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)
7 (PAT)	If the PAT is supported, indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2); otherwise, reserved (must be 0)[1]
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise
11:9	Ignored
31:12	Physical address of the 4-KByte page referenced by this entry

x86 Page Directory Entry (PDE)

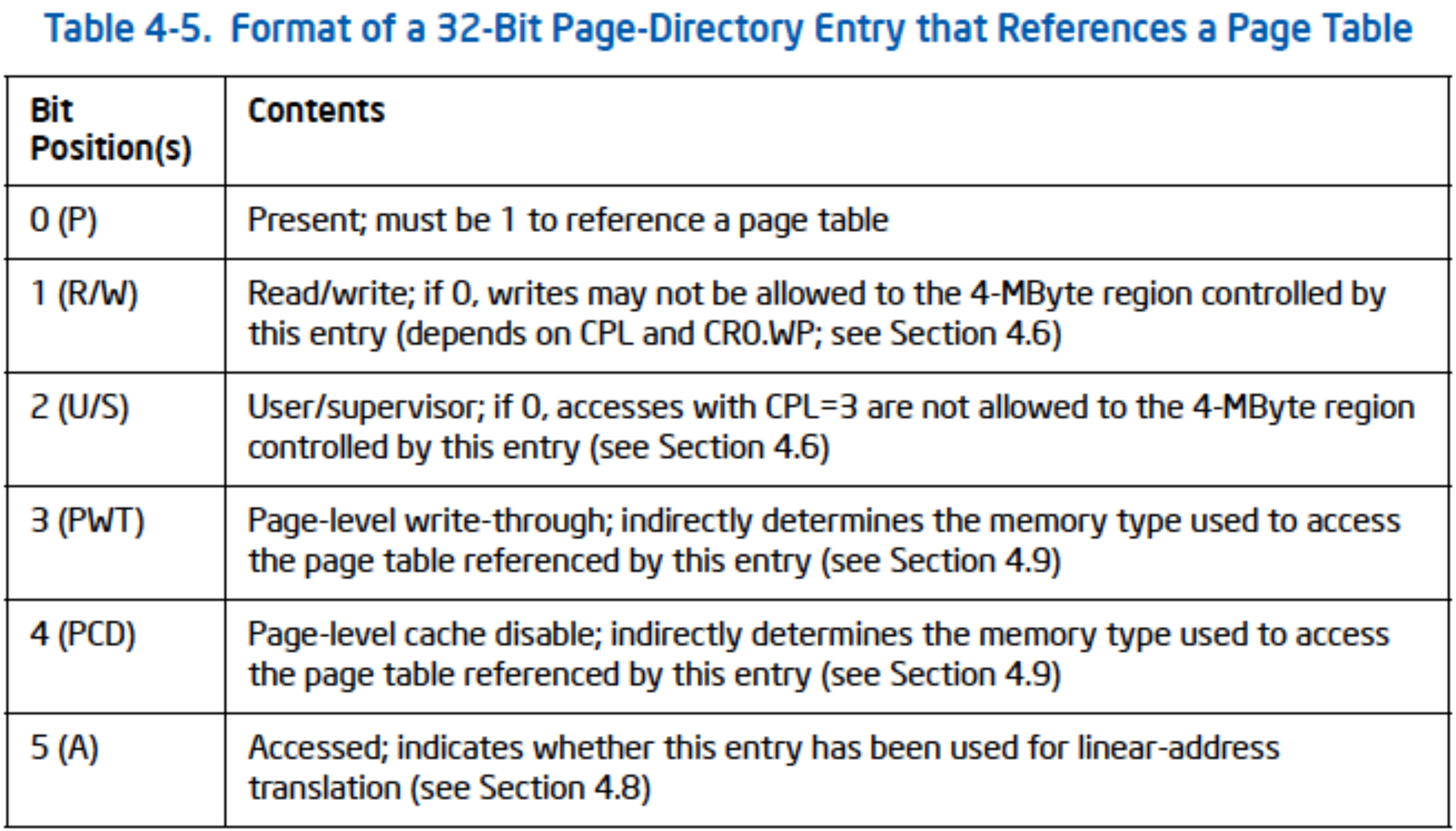


Table 4-5. Format of a 32-Bit Page-Directory Entry that References a Page Table

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-MByte region controlled by this entry (depends on CPL and CR0.WP; see Section 4.6)
2 (U/S)	User/supervisor; if 0, accesses with CPL=3 are not allowed to the 4-MByte region controlled by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)

Four-level Paging in x86

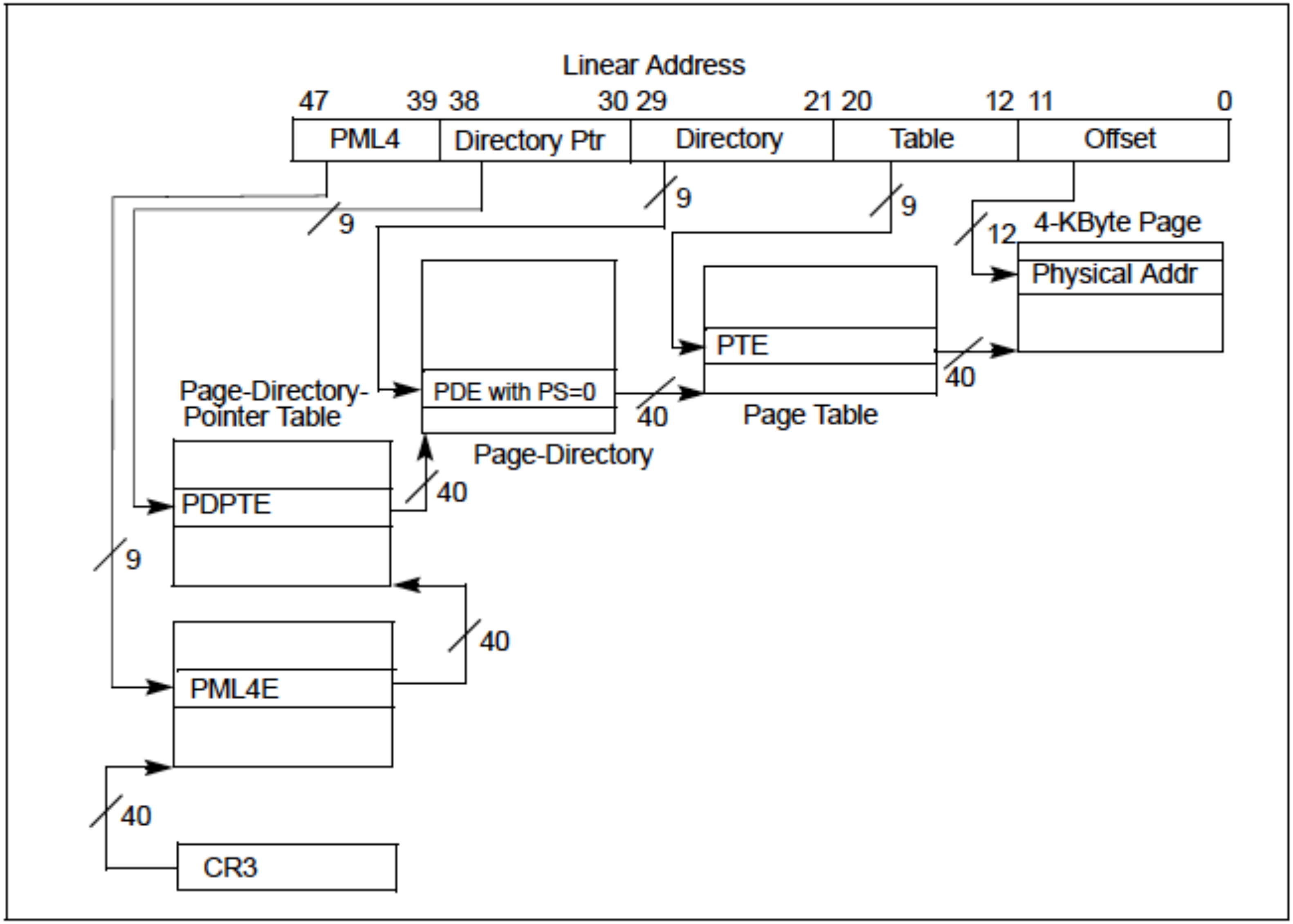
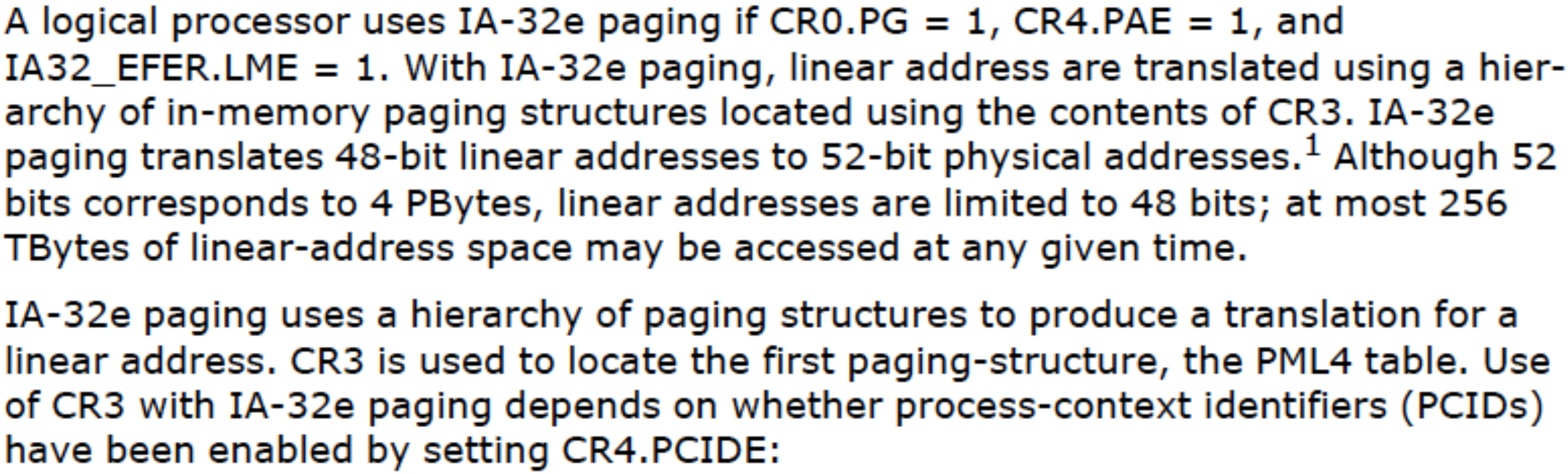


Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

Four-level Paging and Extended Physical Address Space in x86



A logical processor uses IA-32e paging if CR0.PG = 1, CR4.PAE = 1, and IA32_EFER.LME = 1. With IA-32e paging, linear address are translated using a hierarchy of in-memory paging structures located using the contents of CR3. IA-32e paging translates 48-bit linear addresses to 52-bit physical addresses.[1] Although 52 bits corresponds to 4 PBytes, linear addresses are limited to 48 bits; at most 256 TBytes of linear-address space may be accessed at any given time.

IA-32e paging uses a hierarchy of paging structures to produce a translation for a linear address. CR3 is used to locate the first paging-structure, the PML4 table. Use of CR3 with IA-32e paging depends on whether process-context identifiers (PCIDs) have been enabled by setting CR4.PCIDE:

Virtual Memory Issue II

- How fast is the address translation?
  - How can we make it fast?

- Idea: Use a hardware structure that caches PTEs → Translation lookaside buffer

- What should be done on a TLB miss?
  - What TLB entry to replace?
  - Who handles the TLB miss? HW vs. SW?

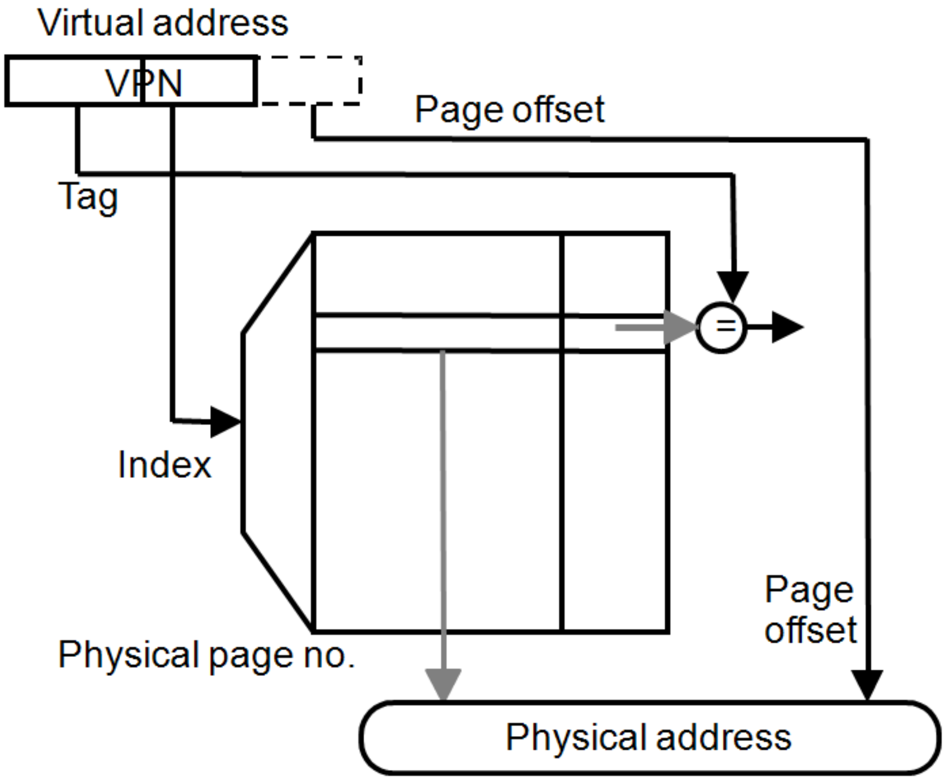
- What should be done on a page fault?
  - What virtual page to replace from physical memory?
  - Who handles the page fault? HW vs. SW?

Speeding up Translation with a TLB

- Essentially a cache of recent address translations
  - Avoids going to the page table on every reference

- **Index** = lower bits of VPN (virtual page #)
- **Tag** = unused bits of VPN + process ID
- **Data** = a page-table entry
- **Status** = valid, dirty

The usual cache design choices (placement, replacement policy, multi-level, etc.) apply here too.



Handling TLB Misses

- The TLB is small; it cannot hold all PTEs
  - Some translations will inevitably miss in the TLB
  - Must access memory to find the appropriate PTE
    - Called **walking** the page directory/table
    - Large performance penalty

- Who handles TLB misses? Hardware or software?

Handling TLB Misses (II)

- Approach #1. **Hardware-Managed** (e.g., x86)
  - The hardware does the **page walk**
  - The hardware fetches the PTE and inserts it into the TLB
    - If the TLB is full, the entry **replaces** another entry
  - Done transparently to system software

- Approach #2. **Software-Managed** (e.g., MIPS)
  - The hardware raises an exception
  - The operating system does the **page walk**
  - The operating system fetches the PTE
  - The operating system inserts/evicts entries in the TLB

Handling TLB Misses (III)

- Hardware-Managed TLB
  - Pro: No exception on TLB miss. Instruction just stalls
  - Pro: Independent instructions may continue
  - Pro: No extra instructions/data brought into caches.
  - Con: Page directory/table organization is etched into the system: OS has little flexibility in deciding these

- Software-Managed TLB
  - Pro: The OS can define page table oganization
  - Pro: More sophisticated TLB replacement policies are possible
  - Con: Need to generate an exception → performance overhead due to pipeline flush, exception handler execution, extra instructions brought to caches

Virtual Memory Issue III

- When do we do the address translation?
  - Before or after accessing the L1 cache?

Virtual Memory and Cache Interaction

Address Translation and Caching

- When do we do the address translation?
  - Before or after accessing the L1 cache?

- In other words, is the cache virtually addressed or physically addressed?
  - Virtual versus physical cache

- What are the issues with a virtually addressed cache?

- Synonym problem:
  - Two different virtual addresses can map to the same physical address → same physical address can be present in multiple locations in the cache → can lead to inconsistency in data

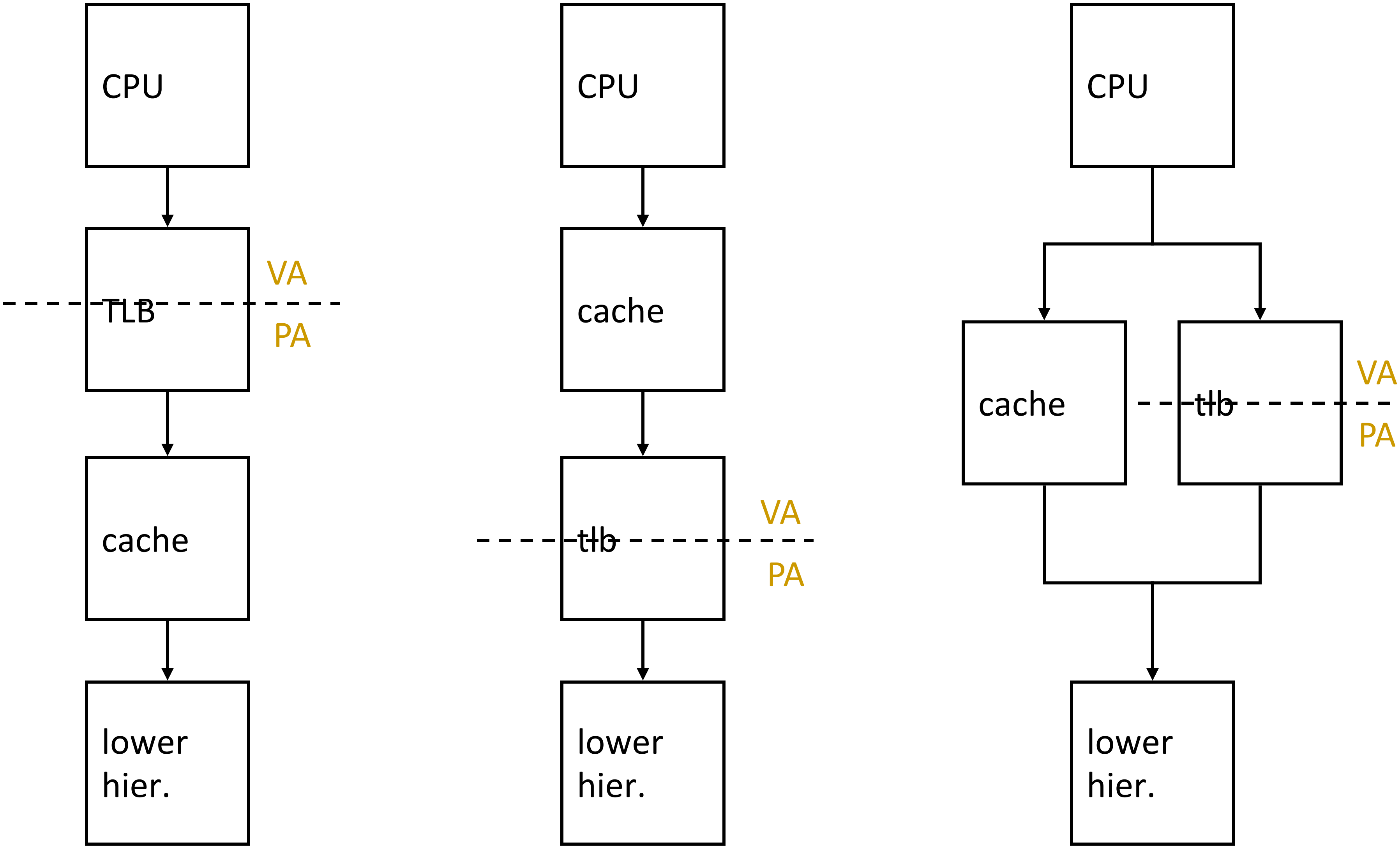
Homonyms and Synonyms

- Homonym: Same VA can map to two different PAs
  - Why?
    - VA is in different processes

- Synonym: Different VAs can map to the same PA
  - Why?
    - Different pages can share the same physical frame within or across processes
    - Reasons: shared libraries, shared data, copy-on-write pages within the same process, …

- Do homonyms and synonyms create problems when we have a cache?
  - Is the cache virtually or physically addressed?

Cache-VM Interaction

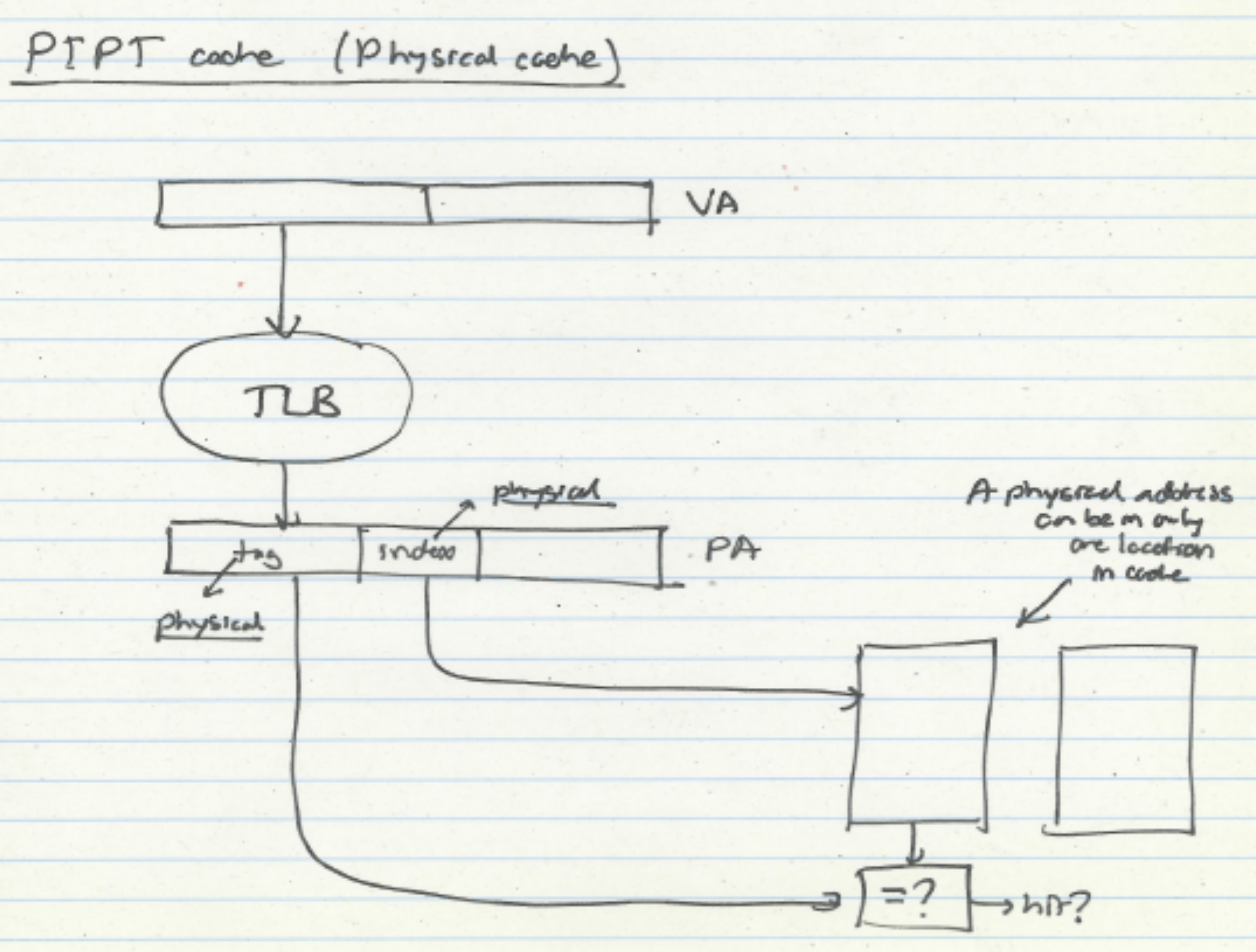


physical cache

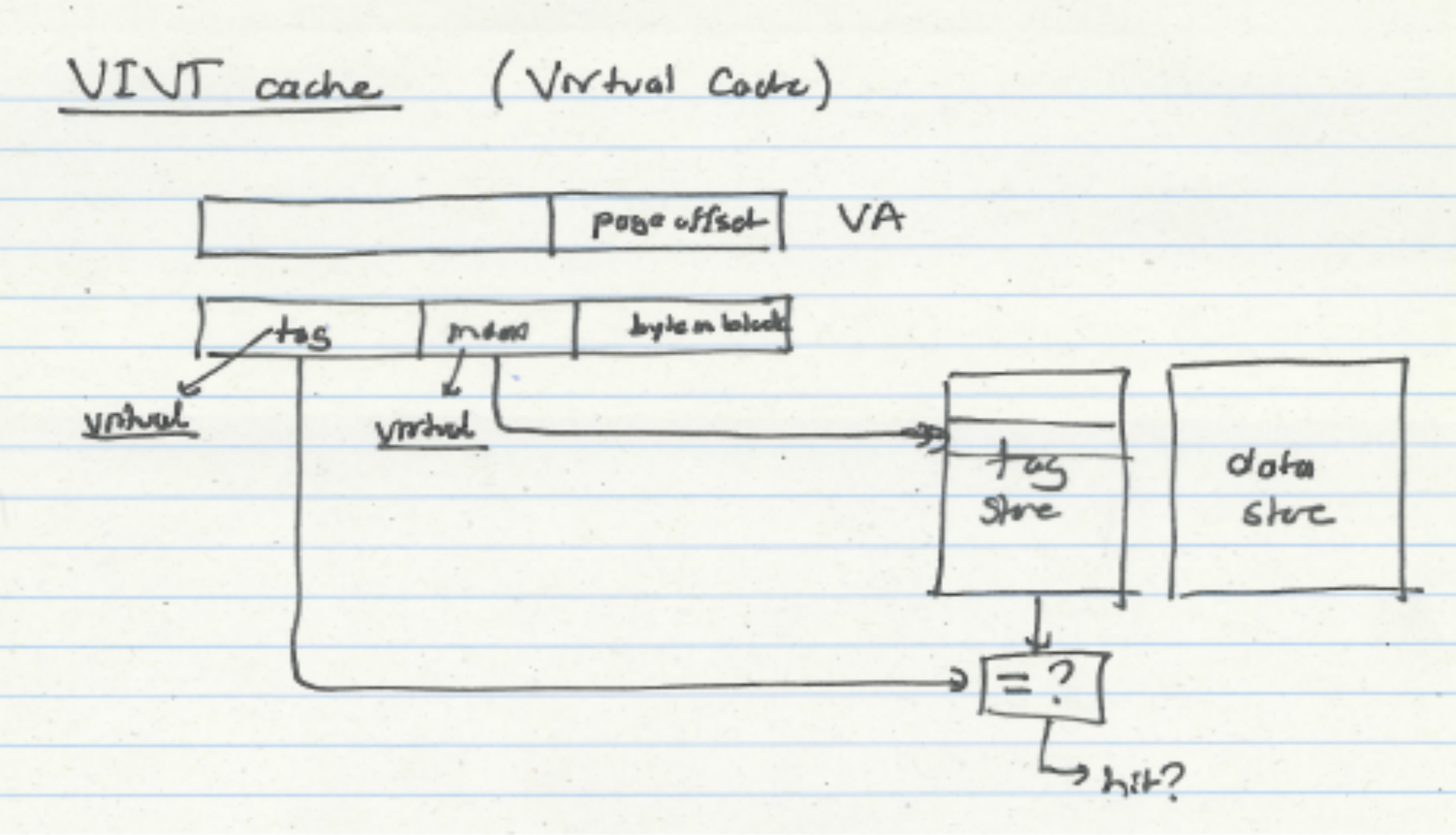
virtual (L1) cache

virtual-physical cache

Physical Cache

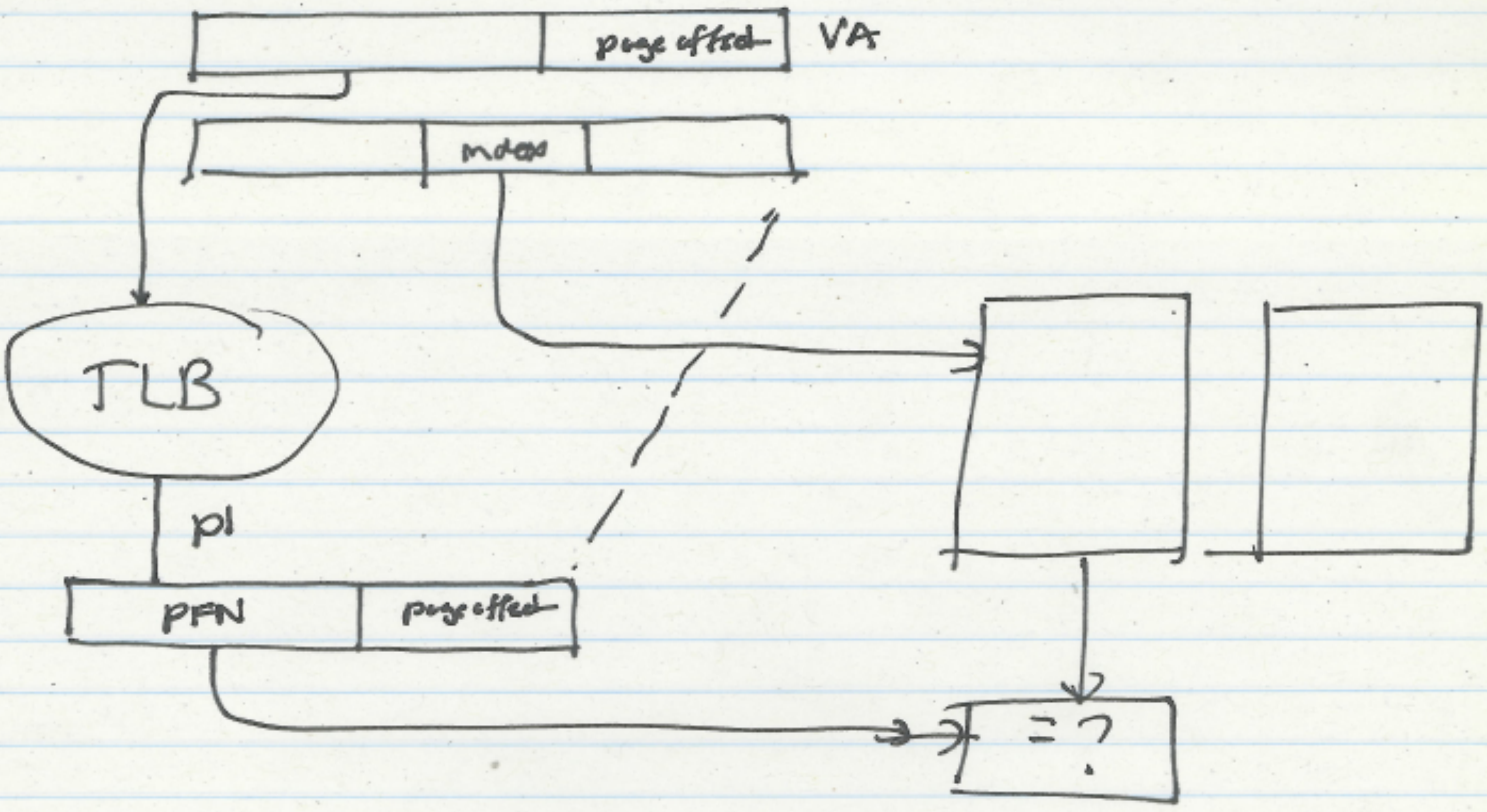

PIPT cache (Physical cache)
VA
TLB
physical
tag
index
PA
physical
A physical address can be in only one location in cache
=?
hit?

Virtual Cache



Virtual-Physical Cache

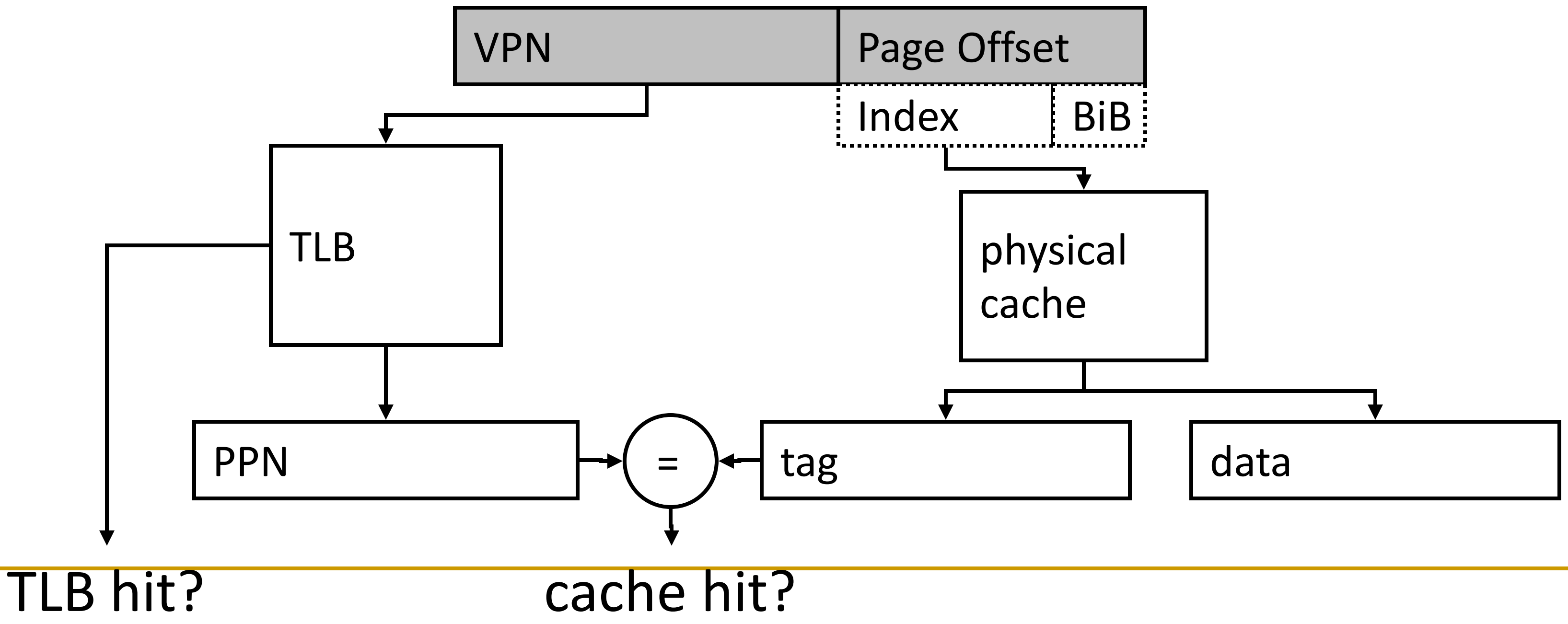
VIPT cache



Where can the same physical address be in the cache?

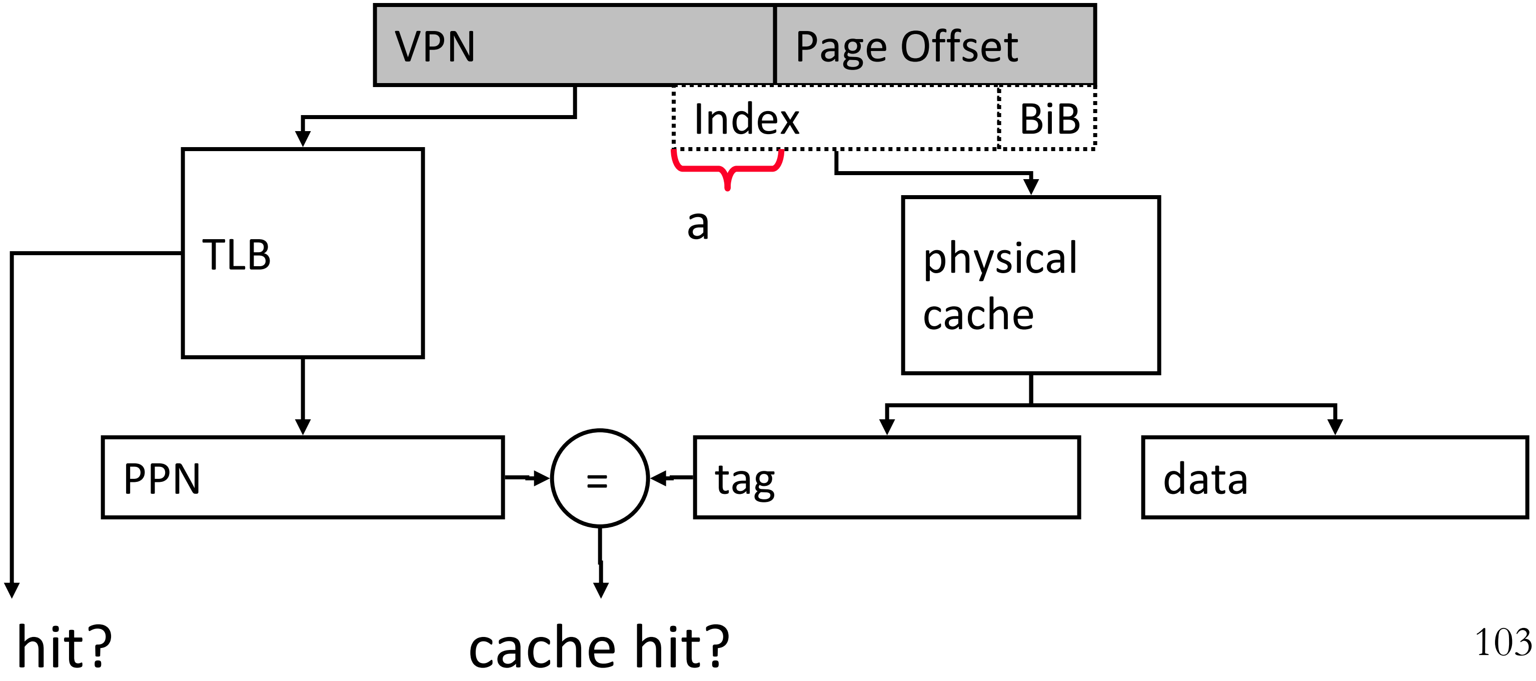
Virtually-Indexed Physically-Tagged

- If C≤(page_size × associativity), the cache index bits come only from page offset (same in VA and PA)
- If both cache and TLB are on chip
  - index both arrays concurrently using VA bits
  - check cache tag (physical) against TLB output at the end



Virtually-Indexed Physically-Tagged

- If C>(page_size × associativity), the cache index bits include VPN
  ⇒ Synonyms can cause problems
  - The same physical address can exist in two locations
- Solutions?



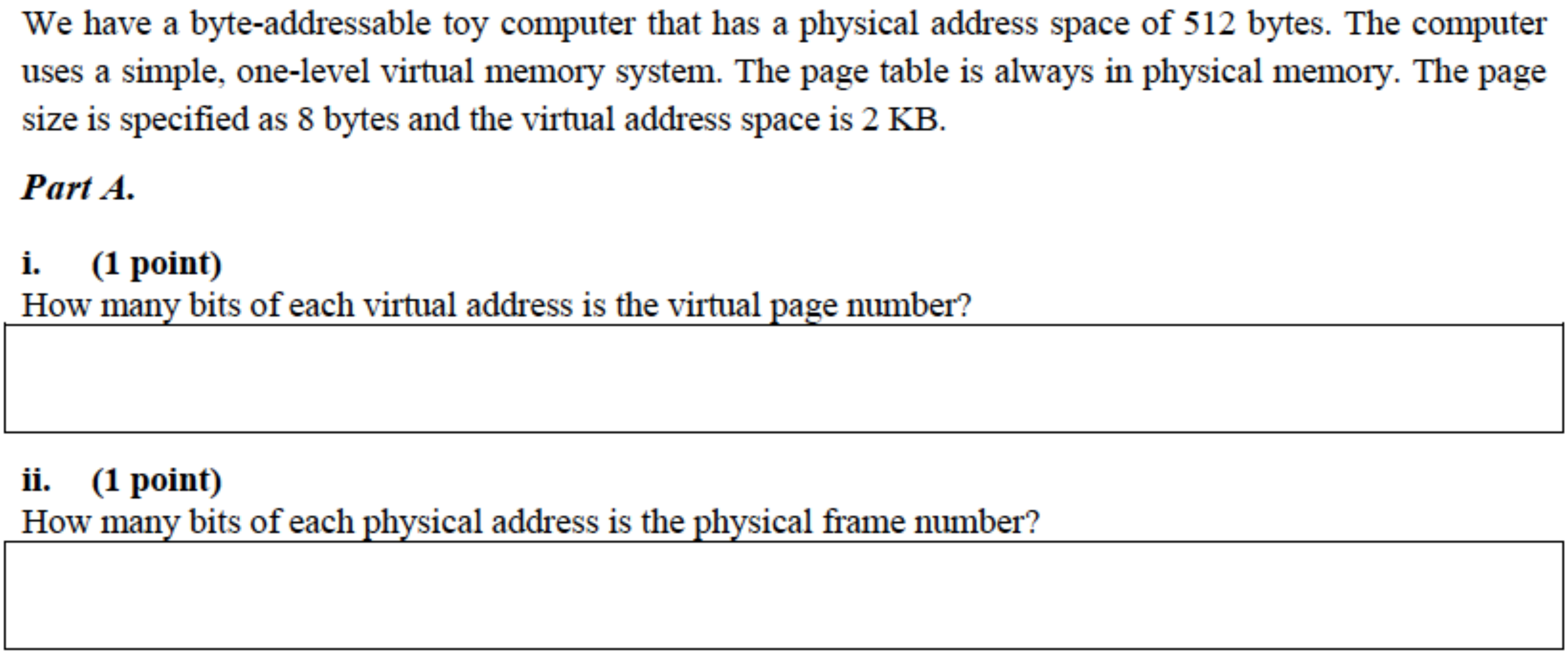
Some Solutions to the Synonym Problem

- Limit cache size to (page size times associativity)
  - get index from page offset

- On a write to a block, search all possible indices that can contain the same physical block, and update/invalidate
  - Used in Alpha 21264, MIPS R10K

- Restrict page placement in OS
  - make sure index(VA) = index(PA)
  - Called page coloring
  - Used in many SPARC processors

An Exercise (I)



We have a byte-addressable toy computer that has a physical address space of 512 bytes. The computer uses a simple, one-level virtual memory system. The page table is always in physical memory. The page size is specified as 8 bytes and the virtual address space is 2 KB.

Part A.

i. (1 point)

How many bits of each virtual address is the virtual page number?

ii. (1 point)

How many bits of each physical address is the physical frame number?

We would like to add a 128-byte *write-through* cache to enhance the performance of this computer. However, we would like the cache access and address translation to be performed simultaneously. In other words, we would like to index our cache using a virtual address, but do the tag comparison using the physical addresses (virtually-indexed physically-tagged). The cache we would like to add is direct-mapped, and has a block size of 2 bytes. The replacement policy is LRU. Answer the following questions:

iii. (1 point)

How many bits of a virtual address are used to determine which byte in a block is accessed?

iv. (2 point)

How many bits of a virtual address are used to index into the cache? Which bits exactly?

v. (1 point)

How many bits of the virtual page number are used to index into the cache?

vi. (5 points)

What is the size of the tag store in bits? Show your work.

Part B.

Suppose we have two processes sharing our toy computer. These processes share some portion of the physical memory. Some of the virtual page-physical frame mappings of each process are given below:

PROCESS 0	
Virtual Page	Physical Frame
Page 0	Frame 0
Page 3	Frame 7
Page 7	Frame 1
Page 15	Frame 3

PROCESS 1	
Virtual Page	Physical Frame
Page 0	Frame 4
Page 1	Frame 5
Page 7	Frame 3
Page 11	Frame 2

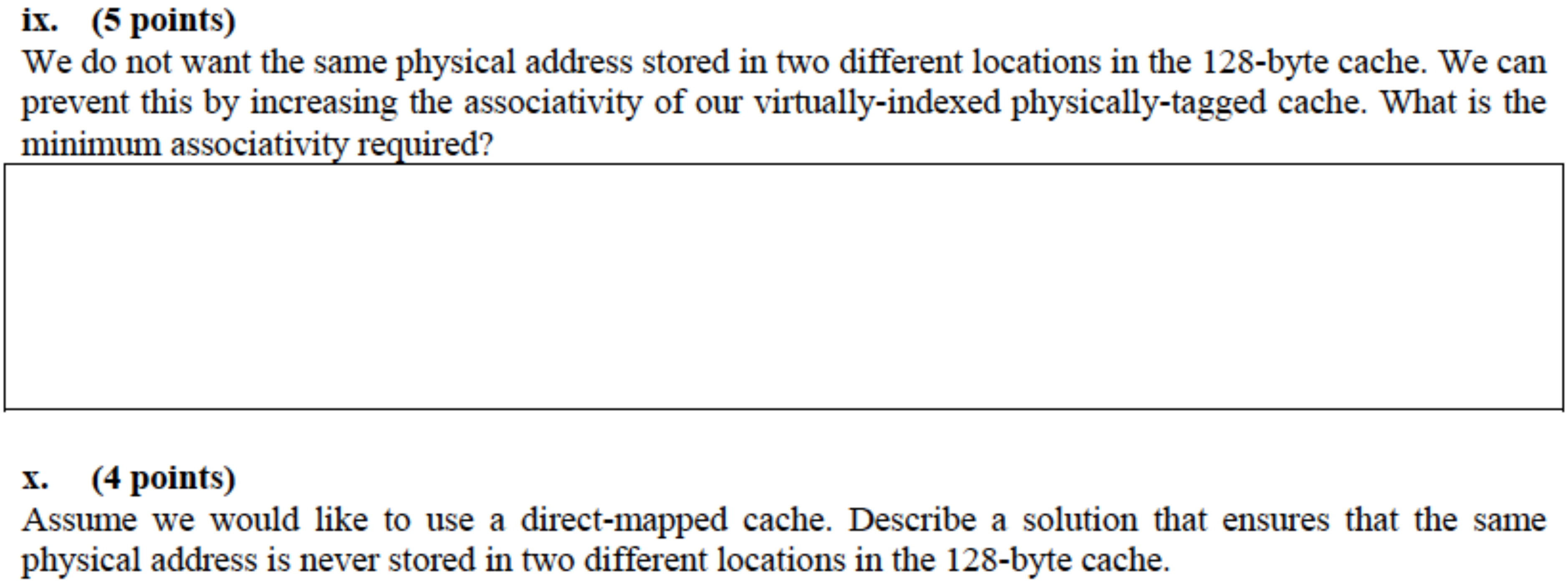
vii. (2 points)

Give a complete physical address whose data can exist in two different locations in the cache.

viii. (3 points)

Give the indexes of those two different locations in the cache.

An Exercise (Concluded)



ix. (5 points)

We do not want the same physical address stored in two different locations in the 128-byte cache. We can prevent this by increasing the associativity of our virtually-indexed physically-tagged cache. What is the minimum associativity required?

x. (4 points)

Assume we would like to use a direct-mapped cache. Describe a solution that ensures that the same physical address is never stored in two different locations in the 128-byte cache.

Some System Software Tasks for VM

- Keeping track of which physical frames are free
- Allocating free physical frames to virtual pages
- Page replacement policy
  - When no physical frame is free, what should be removed?
- Sharing pages between processes
- Copy-on-write optimization
- Page-flip optimization